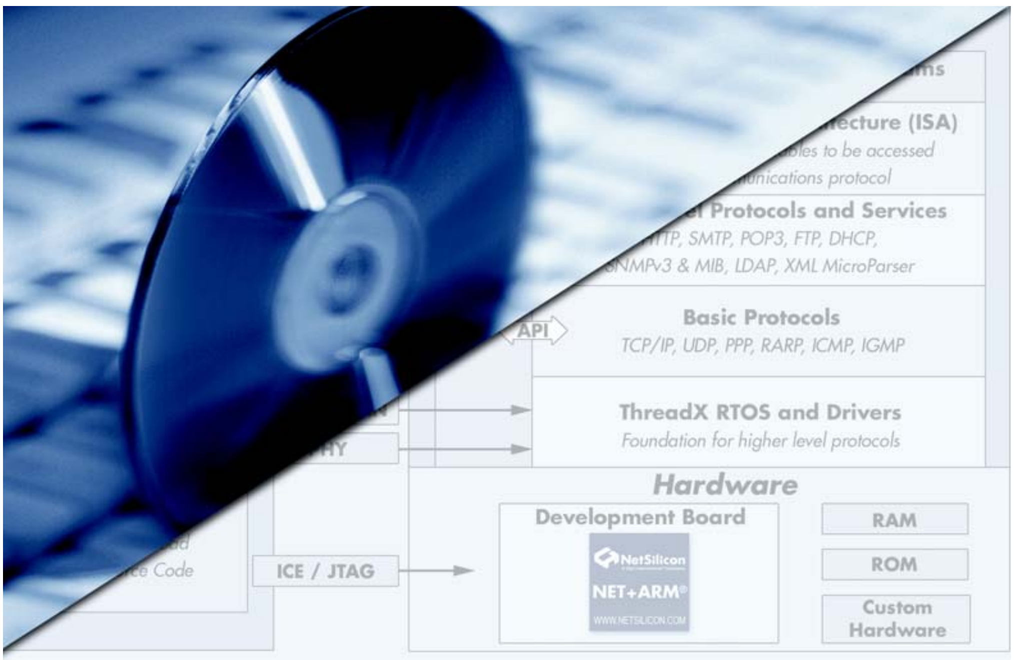
NetSilicon
A Digi International® Company

NET+OS Application Software Reference Guide



NET+OS 5.0
8833443B

NET+OS Application Software Reference Manual

Operating system/version: NET+OS 5.0
Part number/version: 8833443B
Release date: July 2002
www.netsilicon.com

Contents

Using This Guide

About this guide

This guide describes the function calls for NET+OS network software application programming interfaces (APIs).

NET+OS, a network software suite optimized for the NET+ARM chip, is part of the NET+Works integrated product family.

Who should read this guide

This guide is for engineers who are developing NET+Works applications.

To complete the tasks described in this guide, you must:

- Be familiar with programming concepts and techniques, especially for network applications and systems
- Have sufficient system (user) privileges to perform the tasks described
- Have access to a computer system that meets NET+Works hardware and software requirements

What's in this guide

- *Chapter 1* is an overview of the how the various APIs are documented in this book and in other books in the NET+OS documentation set.
- *Chapters 2 – 13* describe the various network software APIs.

Conventions used in this guide

This table describes the typographic conventions used in this guide:

This convention	Is used for
italic type	Emphasis, new terms, variables, and document titles.
`monospaced type`	Filenames, pathnames, commands, and code examples.

Related documentation

- *NET+OS Getting Started Guide* explains how to install NET+OS with Green Hills or with GNU tools, and how to build your first application.
- *NET+OS User's Guide* describes how to use NET+OS to develop programs for your application and hardware.
- *NET+OS BSP Porting Guide* describes how to port the board support package (BSP) to a new hardware application, with either Green Hills Software or GNU tools.
- *NET+OS BSP Software Reference Guide* describes the board support package APIs.
- *NET+OS Kernel User's Guide* describes the real-time NET+OS kernel services.
- Review the documentation CD-ROM that came with your development kit for information on third-party products and other components.
- Refer to the NET+Works hardare documentation for information appropriate to the chip you are using.

Customer support

To get help with a question or technical problem with this product, or to make comments and recommendations about our products or documentation, use the contact information listed in this table:

For	Contact information
Technical support	Telephone: 1 800 243-2333/ 1 781 647-1234 Fax: 1 781 893-1388 Email: tech_support@netsilicon.com
Documentation	techpubs@netsilicon.com
NetSilicon home page	www.netsilicon.com
Online problem reporting	www.netsilicon.com/EmbWeb/Support/forms/bugreport.asp An engineer will analyze the information you provide and call you about the problem.

Introduction to NET+OS Network Software APIs

CHAPTER 1

This guide describes the function calls for the NET+OS network software application programming interfaces (APIs). The APIs described in this guide:

- Implement high-level Internet protocols and services, such as FTP, SNMP, Telnet, and so on
- Provide interfaces to Internet protocols, such as the user interface for systems using the POP3 e-mail protocol

Overview of NET+OS APIs

NET+OS provides three types of APIs, summarized in the following sections.

Core APIs

The core APIs provide access to NET+OS kernel services. They are described in the *NET+OS Kernel User's Guide*.

- Task management for dynamic creation and deletion of tasks and control of task attributes
- Storage allocation of variable size segments and fixed size buffers
- Message queue service for general-purpose communication and synchronization
- Event and asynchronous signal devices
- Semaphore services
- Time management and timer services including maintenance of calendar time and date, timeout and wakeup of tasks, timeslice tracking and round-robin scheduling, and so on

Board support APIs

The board support APIs provide access to some drivers, memory, utilities, and other board and system components. They are described in the *NET+OS BSP Software Reference Guide*.

- Parallel port driver
- Serial port driver
- ENI driver
- LED control
- NVRAM support
- Flash memory support
- Serial number support
- Cache support
- ISR support

- System clock and timer support
- HDLC driver
- DMA driver
- SPI driver
- Serial EEPROM support

Network software APIs

The following table summarizes the network software APIs described in detail in this guide:

Network API type	Description
Advanced Web Server (see Chapter 2)	Provides HTTP service using the Advanced Web Server (AWS).
FTP Server (see Chapter 3)	Provides a way to communicate with FTP clients — for example, enabling an FTP client to download software from a server.
FTP Client (see Chapter 4)	Provides a way to communicate with FTP servers — for example, to transfer data to and from a NET+Works node to a central server.
Domain Name Service (DNS) (see Chapter 5)	Allows applications to access DNS servers to translate DNS names into IP addresses, and vice versa.
E-mail Services (see Chapter 6)	Lets you customize a mail client according to your application's needs.
Simple Network Management Protocol (SNMP) (see Chapter 7)	Enables communications for management and diagnostic infor-mation between agents in the network elements and the network management stations.
Point-to-Point Protocol (PPP) (see Chapter 8)	Enables devices to communicate through a serial line.
Fast IP (see Chapter 9)	Allows applications using Ethernet to bypass some socket processing as a way to get faster response to network device requests.
Dynamic Host Configuration Protocol (DHCP) (see Chapter 10)	Provides for dynamically allocating IP addresses to devices on the network.

Network API type	Description
Telnet (see Chapter 11)	Allows workstations to connect to a remote host and function like terminals hard-wired to the host.
Sockets (see Chapter 12)	Provides the basic building blocks for communication. Tasks communicate by sending and receiving data through sockets.
HTTP Server (see Chapter 13)	Lets you configure Web page access and page properties and define addresses and the page content.
LDAP Services (see Chapter 14)	Lets you use the Lightweight Directory Access Protocol (LDAP).
Management (see Chapter 15)	Lets you access management variables on the system.
General-Purpose and System Access (see Chapter 16)	General-purpose functions system security functions (for setting and getting account information).

About the API descriptions

This guide describes the API function calls using the following format:

- Name of the function and a brief description
- The format of the function call
- A table defining the function’s arguments
- A table defining the function’s return values

In some case, there are additional comments, usage notes, and code examples.

Deprecated functions

Some functions described in this manual are *deprecated* — that is, their use is not recommended, even though the functions may continue to be included for compatibility purposes. Deprecated functions are indicated with a dagger symbol (†).

Private structures

Routines or data structures described in header files but not discussed in this guide, are considered *private structures*. That is, they are *for internal use only* and their functioning is not guaranteed, nor are their prototypes. Using these private structures is strongly discouraged.

Advanced Web Server API

CHAPTER 2

Overview

The Advanced Web Server (AWS) provides HTTP V1.1 service. It differs from the HTTP server interface provided in earlier versions of NET+OS which used the HTML-To-C Converter.

PBuilder support

The Advanced Web Server uses the PBuilder utility to generate three kinds of files:

- Static definitions, including the data structures that represent the Web pages
- Web page stubs for the developer to complete, enabling dynamic data manipulation

- PBuilder utility files (for example, `rppages.c`) which contain common phrases and the data structure containing all the pages served by the server

For more information on PBuilder, see the *Web Application Toolkit PBuilder User's Guide*, provided on the NET+OS documentation CD.

AWS advantages

The advantages of the Advanced Web Server are:

- Web page compression
- File upload capability based on RFC 1867
- File system stubs
- External CGI
- Use of cookies
- HTTP V1.1 compatibility

Moreover, the *comment tags*, described in the *Web Application Toolkit PBuilder User's Guide*, provide easier and more advanced hooks for dynamic data into Web pages.

The AWS interface consists of the Web server startup routine, several stub functions, and the list of Web pages used in the application.

Include files

Using the Advanced Web Server API requires the following header files:

- `http_awsapi.h`
- `http_security.h`
- `maw_api.h` (if you want to create Web pages that access management variables)

For more information on the management API, see Chapter 15.

Summary of Advanced Web Server API functions

Function	Description
RpHSStartServer	Initializes and starts the Web server.
RpHSSetRealm †	Allows an application to set a specified security realm for Web pages.
RpHSGetServerData	Returns a handle to the internal data structure maintained by the server. This handle can then be used in various PBuilder functions.
NAHttpSetRealmSecurity	Enables an application to set the security features on a specified realm.
Stub functions — These functions can be over-written for CGI, file manipulation, or security. Except where indicated otherwise, these functions are in rphttd\file.c	
RpUserExitInit	Initializes user exit resources. Location: rphttd\cgi.c.
RpUserExitDeInit	De-initializes user exit resources. Location: rphttd\cgi.c.
RpExternalCgi	Determines that the URL needs to be handled externally (when control is passed to an external CGI routine).
RpHSOpenFileSystem	Called when the Web server starts up.
RpHSCloseFileSystem	Called when the Web server finishes so that the file system can de-initialize any internal variables and processes.
RpHSCreateFile	Creates a new file on the file system.
RpHSCreateFileStatus	Determines whether the HSCreateFile call has completed.
RpHSCloseFile	Signals the end of the file upload.
RpHSCloseFileStatus	Determines whether a file has been closed.
RpHSWriteFile	Starts to write from the buffer provided for the number of bytes in the count.

Function	Description
RpHSWriteFileStatus	Determines whether the write operation is finished.
RpHSOpenFile	Opens an individual file.
RpHSOpenFileStatis	Determines when the file has been opened.
RpHSReadFile	Starts a read into the buffer provided for the number of bytes in the count.
RpHSReadFileStatus	Determines whether the read operation has finished.
RpHSInitSecurityTable	Initializes all the security data for the server. Location: rphttd\security.c.
Management variables and the Advanced Web Server (MAW)	
mawInstallErrorHandler	Installs an error handler to be used when the application accesses management variables.
mawInstallSubscriptsFunction	Installs a function to provide subscripts to array elements.
mawInstallTimeoutFunction	Installs a timeout function to be used when the application accesses management variables.
mawSetAccessTimeout	Sets the access timeout.

† = Deprecated function

Advanced Web Server API functions

The following pages describe the Advanced Web Server API functions.

RpHSStartServer

Starts the Advanced Web Server task.

Format

```
int RpHSStartServer (void);
```

Arguments

none

Return values

Return value	Description
0	Success
non-zero	Failure

RpHSSetRealm

Allows an application to set one of 8 available realms used for security of Web pages.

Note: Deprecated function. Instead, the system access database should be populated by calling `NAsetSysAccess` (in `sysAccess.h`). Using `RpHSSetRealm` will compile. However, system user account access must be made through the system access database API.

Format

```
int RpHSSetRealm (int realmno, char *realmname,
     char *username, char *password);
```

Arguments

Argument	Description
realmno	Integer specifying the particular realm: 0 – Realm 1 1 – Realm 2 2 – Realm 3 3 – Realm 4 4 – Realm 5 5 – Realm 6 6 – Realm 7 7 – Realm 8
realmname	Pointer to the realm name.
username	Pointer to the user name required for authentication.
password	Pointer to the password required to complete authentication.

Return values

Return value	Description
0	Success
non-zero	Failure

RpHSGetServerData

Returns a handle to the internal data structure maintained by the server. This handle can then be used in various PBuilder functions.

Format

```
void* RpHSGetServerData (void);
```

Arguments

none

Return values

Returns a pointer to a data structure maintained by the Web server to be used in various PBbuilder calls that have a `*theServerDataPtr` argument (see the *Web Application Toolkit PBuilder User's Guide*, provide on the NET+OS CD).

NAHttpSetRealmSecurity

Enables an application to set the security features — access and authentication — on any of 8 available realms.

Format

```
int NAHttpSetRealmSecurity (int realmno, char realmnname,
     int accessType, int authenticationType);
```

Arguments

Argument	Description
realmno	Integer specifying the particular realm: 0 – Realm 1 1 – Realm 2 2 – Realm 3 3 – Realm 4 4 – Realm 5 5 – Realm 6 6 – Realm 7 7 – Realm 8
realmnname	Realm name.
accessType	One of the following: ■ NA_HTTP_SECURITY_MULTIPLE_ACCESS — allows multiple clients to access the realm ■ NA_HTTP_SECURITY_SINGLE_ACCESS — allows only one client to access the realm

Argument	Description
authenticationType	One of the following: ■ NA_HTTP_DIGEST_AUTHENTICATION — uses an MD5 digest over several fields in the packet, including the realm password Transmits the 128-bit digest result instead of the password. ■ NA_HTTP_BASIC_AUTHENTICATION — uses Base64 encoding of the username and password This is less secure but is quickly encoded and works with both Microsoft Internet Explorer and Netscape.

Return values

Return value	Description
0	Success
NA_HTTP_SERVER_NOT_INITIALIZED	Server initialization function was not called before calling this function
NA_HTTP_REALM_NOT_INITIALIZED	Security realm has not been initialized — typically because HSInitSecurityTable or RpHSInitSecurityTable has not yet been called
NA_HTTP_INVALID_REALM	Invalid realm specified Realm number should be an integer (0–7)
NA_HTTP_INVALID_REALMNAME	Null pointer input of *realmname*, or the length of the name is greater than 31
NA_HTTP_INVALID_ACCESS	Invalid access specified
NA_HTTP_INVALID_AUTHENTICATION	Invalid authentication specified

RpUserExitInit

Initializes user exit resources.

Format

```
int RpUserExitInit (void);
```

Arguments

none

Return values

Return value	Description
0	Success
non-zero	Failure

RpUserExitDeInit

De-initializes user exit resources.

Format

```
int RpUserExitDeInit (void)
```

Arguments

none

Return values

Return value	Description
0	Success
non-zero	Failure

RpExternalCgi

Determines that the URL needs to be handled externally (when control is passed to an external CGI routine).

Format

```
void RpExternalCgi(void *theDataPtr, rpCgiPtr theCgiPtr);
```

Arguments

Argument	Description
theDataPtr	Private data structure used to keep information on the internal HTTP server state. (**Note:** This argument should not be used.)
theCgiPtr	Public data structure containing all the pertinent information discussed in the *Usage notes* (below).

Usage notes

The Web server passes control to an external CGI routine by issuing the `RpExternalCgi` call when the server determines that a URL needs to be handled externally. It passes a condensed form of the information in the HTTP request to the external CGI routine using the `rpCgi` control block, and looks for responses from the CGI routine in the same structure.

The arguments passed to the CGI routine include the number of the TCP connection which is passed in `fConnectionId`. The HTTP request type (`GET`, `HEAD`, or `POST`) is passed as an `enum` in `fHttpRequest`. The URL is passed in `fPathPtr` and the contents of various HTTP headers are passed in `fHostPtr`, `fRefererPtr`, `fAgentPtr`, and `fLanguagePtr`. The contents of the Date header are passed in `fBrowserDate` after being converted to the PBuilder internal format of seconds since 1/1/1901. The `fArgumentBufferPtr` and `fArgumentBufferLength` fields point to any query arguments appended to a GET request, or the object body of a POST request. If the browser has provided authentication information, it will be passed in the `fUserNamePtr` and `fPasswordPtr` fields.

The `fResponseState` field is used by the CGI routine to signal its processing state. Since some CGI processes may need to run asynchronously, they can signal this using the `eRpCgiPending` state. PBuilder will issue another `RpExternalCgi` call at a later time to gather the response. If the `eRpCgiLastBuffer` state is returned, the engine knows the CGI process is complete and will send the response back to the browser client. If the `eRpCgiBufferComplete` state is returned, the engine will send the response back to the browser client and issue another `RpExternalCgi` call to gather more of the response.

The `fHttpResponse`, `fDataType`, and `fObjectDate` fields are used to tell the engine which HTTP headers to prepare for the response. The normal response will be `eRpCgiHttpOk` for a dynamically prepared object and `eRpCgiHttpOkStatic` for a static object such as a GIF or JPEG image. The `eRpCgiHttpRedirect` response is used after processing a form to tell the browser which page to retrieve next. The `eRpCgiHttpNotModified` response is used for requests for a static object that have been previously filled. This response can be used to save sending the object to the browser again. The `eRpCgiHttpNotFound` response is used to notify the browser that the CGI routine was not able to handle this request. The `eRpCgiHttpUnauthorized` response is used to tell the browser that the provided User and Password fields are invalid.

The values for `fDataType` are the MIME types specified in `RpMimes.h`. The value of `fObjectDate` is a date in internal PBuilder format if the object is a static object. The `fResponseBufferPtr` and `fResponseBufferLength` point to the HTML response buffer prepared by the external CGI routine. If the `fHttpResponse` field is `eRpCgiHttpRedirect`, the external CGI routine needs to provide the URL to redirect to in the response buffer. If the `fHttpResponse` field is `eRpCgiHttpUnauthorized`, the external CGI routine needs to provide the realm name in the response buffer.

rpCgiPtrStructure

The following structure, in the `RpCgi.h` file, defines the `rpCgiPtr` type used in `theCgiPtr`:

```
typedef struct {
        Unsigned8 fConnectionId;
        rpCgiHttpRequest fHttpRequest;
        char * fPathPtr;            /* URL */
        char * fHostPtr;            /* Host: */
        char * fRefererPtr;         /* Referer: */
        char * fAgentPtr;           /* User-Agent: */
        char * fLanguagePtr;        /* Content-Language: */
        Unsigned32 fBrowserDate;    /* Date: (internal) */
        char * fArgumentBufferPtr;
        Signed32 fArgumentBufferLength;
        char * fUserNamePtr;         /* Username */
        char * fPasswordPtr;         /* Password */
        void * fUserDataPtr;         /* Arbitrary User Data */
        rpCgiResponse fResponseState;
        rpCgiHttpResponse fHttpResponse;
        rpDataType fDataType;
        char * fResponseBufferPtr;
        Signed32 fResponseBufferLength;
        Unsigned32 fObjectDate;         /* internal Object Date */
        Unsigned16 fHostIndex;
} *rpCgiPtr;
```

Note: The `rpCgiHttpRequest`, `rpCgiResponse`, and `rpCgiHttpResponse` enumerations are also in the `RpCgi.h` file.

Return values

none

RpHSOpenFileSystem

Called when the Web server starts up.

Format

```
int RpHSOpenFileSystem (int theOpenFilesCount);
```

Arguments

Argument	Description
theOpenFilesCount	Number of allowed open files.

Usage notes

The maximum number of simultaneous open files is passed in to the file system so that the file system interface can dynamically initialize its internal variables and processes. The number passed in will usually be the same as the number of simultaneous HTTP requests that the Web server engine supports.

Return values

Return value	Description
0	Success
non-zero	Failure

RpHSCloseFileSystem

Called when the Web server finishes, so that the file system can de-initialize any internal variables and processes.

Format

```
int RpHSCloseFileSystem (void);
```

Arguments

none

Return values

Return value	Description
0	Success
non-zero	Failure

RpHSCreateFile

Creates a new file on the file system.

This call is nonblocking and completes when the open operation has been started. Use `RpHSCreateFileStatus` to check for completion.

Format

```
int RpHSCreateFile (int theFilesNumber,
        char *theFullNamePtr, FILEINFOPTR theFileInfoPtr);
```

Arguments

Argument	Description
theFilesNumber	File ID.
theFullNamePtr	Name of the file.
theFileInfoPtr	Pointer to a completed file info block.

The following data structure is defined in `RpExtern.h`:

```
typedef struct
{
   Unsigned32 FileSize; /* size of the file in bytes */
   Unsigned32 FileDate; /* date of the file in seconds
                           since January 1, 1901 */
   rpDataType FileType; /* mime type */
   Unsigned8 FileAccess
   void /*UserData;
} FILEINFO, *FILEINFOPTR;
```

Data contents are described in `rpDataType` (also defined in the `RpExtenr.h`) .

Return values

Return value	Description
0	Success
non-zero	Failure

RpHSCreateFileStatus

Determines whether the create operation started in the `RpHSCreateFile` call has completed.

Format

```
int RpHSCreateFileStatus (int theFilesNumber,
        *theCompletionStatusPtr);
```

Arguments

Argument	Description
theFilesNumber	File ID.
theCompletionStatusPtr	Pointer to the call status: ▪ `ACTIONCOMPLETED` — file has been created ▪ `ACTIONPENDING` — file is not open yet

Return values

Return value	Description
0	Success
non-zero	Failure

RpHSCloseFile

Signals the end of file upload.

This call is nonblocking and completes when the close operation has been started. Use `RpHSCloseFileStatus` to check for completion.

Format

```
int RpHSCloseFile (int theFilesNumber,
        int theCompleteFlag);
```

Arguments

Argument	Description
theFilesNumber	File ID.
theCompleteFlag	Signals whether all data was received from the browser: ▪ 1 — file is complete ▪ 0 — file is incomplete

Return values

Return value	Description
0	Success
non-zero	Failure

RpHSCloseFileStatus

Determines whether the close operation started in the `RpHSCloseFile` call has completed.

Format

```
int RpHSCloseFileStatus (int theFilesNumber,
        int *theCompletionStatusPtr);
```

Arguments

Argument	Description
theFilesNumber	File ID.
theCompletionStatusPtr	Pointer to the call status: ▪ `ACTIONCOMPLETED` — file has been closed ▪ `ACTIONPENDING` — file is still open

Return values

Return value	Description
0	Success
non-zero	Failure

RpHSWriteFile

Starts a write operation from the buffer provided for the number of bytes in the count.

Use `RpHSWriteFileStatus` to check for completion.

Format

```
int RpHSWriteFile (int theFilesNumber, char *theWritePtr,
       unsigned long theByteCount);
```

Arguments

Argument	Description
theFilesNumber	File ID.
theWritePtr	Pointer to the write buffer.
theByteCount	Number of bytes to write.

Return values

Return value	Description
0	Success
non-zero	Failure

RpHSWriteFileStatus

Determines whether the write operation started in the `RpHSWriteFile` call has completed.

Format

```
int RpHSWriteFileStatus (int theFilesNumber,
        int *theCompletionStatusPtr,
        unsigned long *theBytesWrittenPtr);
```

Arguments

Argument	Description
theFilesNumber	File ID.
theCompletionStatusPtr	Pointer to the call status: ■ `ACTIONCOMPLETED` — the file has been written ■ `ACTIONPENDING` — the file is still not written
theBytesWrittenPtr	Pointer to length field. On return, the number of bytes actually written is stored in the caller’s length field.

Return values

Return value	Description
0	Success
non-zero	Failure

RpHSOpenFile

Opens an individual file.

Format

```
int RpHSOpenFile (int theFilesNumber,
        char *theFullNamePtr);
```

Arguments

Argument	Description
theFilesNumber	File ID.
theFullNamePtr	Pointer to the full URL object name.

Usage notes

The file byte position is set to 0. The open file call is responsible for all directory positioning since the full file name from the URL will be passed in. This call is nonblocking and completes when the open operation has been started. Use `RpHSOpenFileStatus` to check for completion. An example file name is:

/directory/subdirectory/`filename.txt`

Return values

Return value	Description
0	Success
non-zero	Failure

RpHSOpenFileStatus

Determines whether the open operation started during the `RpHSOpenFile` call has completed.

Format

```
int RpHSOpenFileStatus (int theFilesNumber,
        int *theCompletionStatusPtr,
        FILEINFOPTR theFileInfoPtr);
```

Arguments

Argument	Description
theFilesNumber	File ID.
theCompletionStatusPtr	Pointer to the call status: ■ `ACTIONCOMPLETED` — file has been opened ■ `ACTIONPENDING` — file is not open yet
theFileInfoPtr	Pointer to an empty file info block.

Return values

Return value	Description
0	Success
`FILENOTFOUND`	Cannot find this file to open
`FILENOTOPEN`	Cannot open this file

RpHSReadFile

Starts a read operation into the buffer provided for the number of bytes in the count.

The read takes place at the current file byte position. The file byte position is updated after the read operation completes. Use `RpHSOpenFileStatus` to check for completion.

Format

```
int RpHSReadFile(int theFilesNumber, char *theReadPtr,
        unsigned long theByteCount);
```

Arguments

Argument	Description
theFilesNumber	File ID.
theReadPtr	Pointer to the read buffer.
theByteCount	Number of bytes to read.

Return values

Return value	Description
0	Success
non-zero	Failure

RpHSReadFileStatus

Determines whether the read started during the `RpHSReadFile` call has completed.

Format

```
int RpHSReadFileStatus(int theFilesNumber,
        int *theCompletionStatusPtr,
        unsigned long *theBytesReadPtr);
```

Arguments

Argument	Description
theFilesNumber	File ID.
theCompletionStatusPtr	Pointer to the call status: ■ 0 — `READPENDING` – read is outstanding ■ 1 — `READCOMPLETE` – file has been read ■ 2 — `ENDOFFILE` – file has been read and there are no more bytes to be read
theBytesReadPtr	Pointer to length field. On return, the number of bytes actually written is stored in the caller's length field.

Return values

Return value	Description
0	Success
non-zero	Failure

RpHSInitSecurityTable

Initializes the security data for the server.

Security calls define security realms, along with the realm's name, access, and authentication type (see `NAHttpSetRealmSecurity`).

Format

```
void RpHSInitSecurityTable (void *severdata);
```

Arguments

The input argument is private and should not be used or modified.

Example

The following line of code can be added to the stub routine to set Realm 1 to "NetSilicon Apps" with single-user access (only one user can access the realm at a time) and MD5 digest authentication:

```
NAHttpSetRealmSecurity (0, "NetSilicon App",
    NA_HTTP_SECURITY_SINGLE-ACCESS,
    NH_HTTP_SECURITY_DIGEST_AUTHENTICATION);
```

Return values

none

mawInstallErrorHandler

Installs an application-supplied error handler which will be called if there is an error in accessing a management variable.

The installed error handler should then either halt the system or handle the error by generating a suitable response that can be passed to the server. The installed function should return a pointer to the response that is to be passed to the server, or should not return at all.

Format

`void mawInstallErrorHandler (mawErrorFn appFunction);`

Arguments

Argument	Description
appFunction	Pointer to the function that will handle errors. To remove (de-install) an error handler, call `mawInstallErrorHandler` with *appFunction* set to `NULL`.

The `mawErrorFn` type is defined as follows:

```
typedef void * (*mawErrorFn) (char *varName,
       AwsDataType htmlType, MAW_ERROR_TYPE error);
```

where:

varName	Name of the variable being accessed
htmlType	Data type expected by the server
error	Error condition

Return values

none

mawInstallSubscriptsFunction

Installs an application-supplied function to provide subscripts to array elements.

The installed function will be called when the application accesses a management variable in array. If the variable is a one-dimensional character array, the subscripts function can return a `NULL` pointer to indicate the entire array should be read or written. Otherwise, the function should return a pointer to an integer array containing the subscripts.

Format

```
void mawInstallSubscriptsFunction (mawSubscriptsFn appFunction);
```

Arguments

Argument	Description
appFunction	Pointer to the function that provides subscripts for array variables. To remove (de-install) a subscripts function, call `mawInstallSubscriptsFunction` with *appFunction* set to `NULL`.

The `mawSubscriptsFn` type is defined as follows:

```
    typedef int *(*mawSubscriptsFn)(char *varName,
        INT16 *indices, int *dimensions,
        int numberDimensions, AwsDataType htmlType);
```

where:

varName	Specifies the variable being accessed
indices	Specify a pointer to an array of integers. These are the current loop indices used by the server
dimensions	A pointer to an array of integers specifying the dimensions of the management variable
numberDimensions	Specifies the number of dimensions in the variable
htmlType	Specifies the data type expected by the server

Return values

none

mawInstallTimeoutFunction

Installs an application-supplied function that will generate appropriate timeout values to use when accessing management variables.

Some management variables may be protected by semaphores. The timeout specifies how many system ticks to wait for the semaphores to unlock before giving up. The installed function will be called every time a management variable is accessed, whether semaphores protect it or not. It will be passed the name of the variable being accessed, and should return the timeout for the variable.

Timeouts returned by the installed function override values set by `mawSetAccessTimeout`.

Format

```
void mawInstallTimeoutFunction (mawTimeoutFn appFunction);
```

Arguments

Argument	Description
appFunction	Pointer to the function that generates timeouts. To remove (de-install) a timeout function, call `mawInstallTimeoutFunction` with *appFunction* set to `NULL`.

The `mawTimeoutFn` type is defined as follows:

```
typedef MAN_TIMEOUT_TYPE (*mawTimeoutFn)(char *varName);
```

where *varName* specifies the name of the variable being accessed.

Return values

none

mawSetAccessTimeout

Sets the access timeout. If management variable is protected by semaphores, the access timeout is the maximum amount of time to wait for the semaphores to unlock before giving up.

Any value set by this function will be ignored if you used `mawInstallTimeoutFunction` to install an application-specific function to generate timeouts.

Format

```
void mawSetAccessTimeout (MAN_TIMEOUT_TYPE timeout);
```

Arguments

Argument	Description
timeout	Number of ticks to wait for semaphores protecting a management variable to unlock.

Return values

none

Example

The following code sets the access timeout:

```
#define ACCESS_TIMEOUT (5*ONE_SECOND)
mawSetAccessTimeout (ACCESS_TIMEOUT);
```

Public structures

The following public structures are defined in files created by the PBuilder utility.

gRpMasterObjectList

An array of structures representing all the Advanced Web Server pages. This structure is created by PBuilder.

Format

```
rpObjectDescPtrPtr gRpMasterObjectList[];
```

Source file

```
rphttd\Rpages.c
```

gUserPhrasesEnglish

An array of common phrases used for compression by PBuilder.

These phrases (definitions) should be defined in the `RpUsrDct.txt` file which is used to generate `RpUsrDct.h` and `RpUsrDct.c`.

Format

```
char *gUserPhrasesEnglish[];
```

Source file

```
rphttd\RpUsrDct.c
```

FTP Server API

C H A P T E R 3

Overview

The FTP server API provides the mechanisms for communicating with FTP clients. It enables NET+Works applications to customize a server according to an application's needs, including letting the application take control based on the FTP command being executed at the server.

For example, when the FTP client puts a file on the server, the server interprets the initial FTP protocol dialog and dispatches it to the application when the file is transferred to the server. To incorporate FTP server capability into a product, you need only implement routines for the necessary commands.

Include file

Using the FTP server API requires the following header file:

```
fservapi.h
```

Summary of FTP server API functions

Function	Description
FSHandleToSocket	Enables FTP server application to retrieve the data socket for an ongoing client connection.
FSInitialize	Initializes the FTP data structures.
FSProperties	Modifies the task characteristics of the FTP server.
FSRegisterControlClose	Registers a callback routine when a control connection is closed.
FSRegisterCWD	Registers the user application routine for changing the working directory.
FSRegisterDataClose	Registers a callback routine when a data connection is closed.
FSRegisterDELE	Registers user application routines for an FTP DELE command.
FSRegisterLIST	Registers the user application routine for listing all the files in the working directory.
FSRegisterMKD	Registers user application routines for an FTP MKD command.
FSRegisterNLST	Registers the user application NLST routine.
FSRegisterRMD	Registers user application routines for an FTP RMD command.
FSRegisterRETR	Registers user application routines for an FTP RETR command.
FSRegisterSTOR	Registers user application routines for an FTP STOR command.
FSRegisterSYST	Registers the user application for an FTP SYST command.
FSRegisterValidation†	Registers a callback routine that validates the username and password.
FSSequenceNumber	Returns the current sequence number of the data socket.

Function	Description
FSStartServer	Starts the server.
FSTimeout	Changes the FTP server inactivity timeout.
FSUserName	Points to the NULL-terminated string containing the username of the current session.

† = Deprecated function.

Sample applications

Sample applications for the FTP server API include:

- Uploading configuration files on an end-user workstation using standard software
- Transferring data to a device's memory or to an internal disk drive.

  For example, a scanner could store scanned images on its hard disk or in memory and allow end users to download images using FTP.

Memory usage

The FTP server currently uses one task to perform all of its functions. The stack size of the task is controlled by the application calling the routine for the server. The server control data structure uses approximately 300 bytes, along with 200 bytes for each user connection. The number of concurrent user connections supported by the server is set by the controlling application.

FTP server API functions

The following pages describe the FTP server API functions.

FSHandleToSocket

Enables an FTP server application to retrieve the data socket for an ongoing client connection.

Typically, the retrieve (`RETR`) routine (defined with `FSRegisterRETR`) for a user application uses `FSHandletoSocket` to get a data socket so that large data packets can be returned in one call. You should use this call within your retrieve routine.

Format

```
int FSHandleToSocket (unsigned long handle);
```

Arguments

Argument	Description
handle	Identifies the network connection used for the request.

Return values

Return value	Description
0 or greater	Indicates a valid data socket
-1	Handle does not represent a valid session

FSInitialize

Initializes the FTP data structures.

Must be called before you call any other FTP server API routines.

Format

```
int FSInitialize (int numusers);
```

Arguments

Argument	Description
numusers	The number of concurrent sessions supported by the server.

Return values

Return value	Description
0	Success — the data structures were properly initialized
-1	Error occurred

FSProperties

Modifies the task characteristics of the FTP server.

Must be called after `FSInitialize` and before `FSStartServer`.

The default settings are sufficient for most applications.

Format

```
void FSProperties (char *tnamep, int priority, int sysstack
       int usrstack, int mode, int flags);
```

Arguments

Argument	Description
tnamep	Pointer to the name of FTP server task.
priority	FTP server task priority. This value depends on the host operating system.
sysstack	Size of the FTP server supervisor stack.
usrstack	Size of the FTP server user stack. Not used in NET+OS implementation of the FTP server.
mode	Aditional task startup information for the server. This value depends on the host operating system requirements.
flags	Additional task startup information for the server. This value depends on the host operating system requirements. Not used in NET+OS implementation of the FTP server.

Return Values

none

FSRegisterControlClose

Registers a callback routine when a control connection is closed.

This is used by applications that require notification when an FTP session is terminated.

Format

```
int FSRegisterControlClose (int (*app_cResetp)
       (char *username, unsigned long handle));
```

Arguments

Argument	Description
app_cResetp	Pointer to the user application for the control close routine.
username	Pointer to a username string that was used to open the control connection.
handle	Handle required for other FTP server API routines.

Return values

Return value	Description
0	Success
-1	Server has already been started, registration has been ignored

FSRegisterCWD

Changes the working directory.

This routine is called when the server receives an FTP `CWD` command.

Format

```
int FSRegisterCWD (int (*app_CWDp) (char *argp,
       unsigned long handle));
```

Arguments

Argument	Description
app_CWDp	Pointer to the user application routine for the `CWD` command.
argp	Pointer to a `NULL`-terminated string containing command parameters.
handle	Handle required for other FTP server API routines.

Return values

Return value	Description
0	Success
-1	Server has already been started, registration has been ignored
Return values for app_CWDp	
0	Command finished The server sends the appropriate FTP completion code to the client to signal success.
-1	Ccommand finished with an error The server sends the appropriate FTP completion code to the client to signal failure.

FSRegisterDataClose

Registers a callback routine for when a data connection is closed.

This is used by applications that require notification when an FTP data connection is terminated.

Format

```
int FSRegisterDataClose (int (*app_dResetp)
       (char *filep, unsigned long handle));
```

Arguments

Argument	Description
app_dResetp	Pointer to the user application data close routine.
filep	Pointer to the username in the current session.
handle	Handle required for other FTP server API routines.

Return values

Return value	Description
0	Success
-1	Server has already been started, registration has been ignored

FSRegisterDELE

Registers a user application callback routine that is called whenever the server receives an FTP `DELE` command.

The callback routine should delete the argument file.

Format

```
int FSRegisterDELE (int (*app_DELEp)(char* argp,
        unsigned long handle));
```

Arguments

Argument	Description
app_DELEp	Pointer to a user application for the `DELE` command.
argp	Pointer to a `NULL`-terminated string containing command parameters.
handle	Handle required for other FTP server API routines

Return values

Return value	Description
0	Success
-1	Server has already been started, registration has been ignored
For app_DELEp	
0	Ccommand finished server sends the appropriate FTP completion code to the client to signal success.
-1	Ccommand finished with an error The server sends the appropriate FTP completion code to the client to signal failure.

FSRegisterLIST

Registers a user application callback routine that is called whenever the server receives an FTP `LIST` command.

The callback routine should list all the file information (for example, date/time stamp, size) in the working directory.

Format

```
int FSRegisterLIST (int (*app_LISTp) (char *argp,
        unsigned long handle));
```

Arguments

Argument	Description
app_LISTp	Pointer to the user application routine for the `LIST` command.
argp	Pointer to a `NULL`-terminated string containing command parameters.
handle	Handle required for other FTP server API routines.

Return values

Return value	Description
0	Success
-1	Server has already been started, registration has been ignored
For app_LISTp	
0	Command finished The server sends the appropriate FTP completion code to the client to signal success.
-1	Command finished with an error The server sends the appropriate FTP completion code to the client to signal failure.

FSRegisterMKD

Registers a user application callback routine that is called whenever the server receives an FTP `MKD` command.

The callback routine should create a directory defined by the argument.

Format

```
int FSRegisterMKD (int (*app_MKDp) (char* argp,
        unsigned long handle));
```

Arguments

Argument	Description
app_MKDp	Pointer to the user application routine for the `MKD` command.
argp	Pointer to a `NULL`-terminated string containing command parameters.
handle	Handle required for other FTP server API routines.

Return values

Return value	Description
0	Success
-1	Server has already been started, registration has been ignored
For app_MKDp	
0	Command finished The server sends the appropriate FTP completion code to the client to signal success.
-1	Command finished with an error The server sends the appropriate FTP completion code to the client to signal failure.

FSRegisterNLST

Registers a user application callback routine that is called whenever the server receives an FTP NLST command.

The callback routine should return the filename information in a readable format for the user.

Format

```
int FSRegisterNLST(int (*app_NLSTp)(char *argp,
        unsigned long handle));
```

Arguments

Argument	Description
app_NLSTp	Pointer to the user application routine for the NLST command.
argp	Pointer to a NULL-terminated string containing command parameters.
handle	Handle required for other FTP server API routines.

Return values

Return value	Description
0	Success
-1	Server has already been started, registration has been ignored
For app_NLSTp	
0	Command finished The server sends the appropriate FTP completion code to the client to signal success.
-1	Command finished with an error The server sends the appropriate FTP completion code to the client to signal failure.

FSRegisterRETR

Registers a user application callback routine that is called whenever the server receives an FTP RETR command.

The callback routine should return the contents of the file.

Format

```
int FSRegisterRETR (int (*app_RETRp)(char* argp,
        unsigned long handle));
```

Arguments

Argument	Description
app_RETRp	Pointer to the user application routine for the RETR command.
argp	Pointer to a NULL-terminated string containing command parameters.
handle	Handle required for other FTP server API routines.

Return values

Return value	Description
0	Success
-1	Server has already been started, registration has been ignored
For app_RETRp	
0	Command finished The server sends the appropriate FTP completion code to the client to signal success.
1	Ccommand has not yet finished The server continues to call the user application routine until the command finishes.

FSRegisterRMD

Registers a user application callback routine that is called whenever the server receives an FTP RMD command.

The callback routine should remove the directory defined by the function argument.

Format

```
int FSRegisterRMD (int (*app_RMDp)(char* argp,
        unsigned long handle));
```

Arguments

Argument	Description
app_RMDp	Pointer to the user application routine for the RMD command.
argp	Pointer to a NULL-terminated string containing command parameters.
handle	Handle required for other FTP server API routines.

Return values

Return value	Description
0	Success
-1	Server has already been started, registration has been ignored
For app_RMDp	
0	Command finished The server sends the appropriate FTP completion code to the client to signal success.
-1	Command finished with an error The server sends the appropriate FTP completion code to the client to signal failure.

FSRegisterSTOR

Registers a user application callback routine that is called whenever the server receives an FTP STOR command. (In an FTP client program, STOR is the PUT operation.)

Format

```
int FSRegisterSTOR (int (*app_STORp)(char *bufferp,
        int buflen, char *argp, unsigned long handle));
```

Arguments

Argument	Description
app_STORp	Pointer to the user application routine for the STOR command.
bufferp	Pointer to the data sent by the FTP client to the server.
buflen	Length of the data.
argp	Pointer to a NULL-terminated string containing command parameters.
handle	Handle required for other FTP server API routines.

Return values

Return value	Description
0	Success
-1	Server has already been started, registration has been ignored
For app_STORp	
0	Success
-1	Connection is closed — transfer aborted due to some error
-2	File size is too large and exceeds the memory size

FSRegisterSYST

Registers a user application callback routine that is called whenever the server receives an FTP SYST command.

Format

```
int FSRegisterSYST (int (*app_SYSTp)(char *argp,
        char *infop, unsigned long handle));
```

Arguments

Argument	Description
app_SYSTp	Pointer to the user application routine for the SYST command.
argp	Pointer to a NULL-terminated string containing command parameters.
infop	Pointer to a 256-character buffer that holds the information returned by the user application. The server returns this information to the requesting client.
handle	Handle required for other FTP server API routines.

Return values

Return value	Description
0	Success
-1	Server has already been started, registration has been ignored

FSRegisterValidation

Registers a callback routine that validates usernames and passwords.

Note: Deprecated function. Instead, the system access database should be populated by calling `NAsetSysAccess` (in `sysAccess.h`). If you use `FSRegisterValidation`, it will compile. However, system user account access must be made through the system access database API.

Format

```
int FSRegisterValidation (int (*app_validp)(char *username,
       char *password));
```

Arguments

Argument	Description
app_validp	Pointer to the user application routine for validating usernames and passwords.
username	Pointer to the username supplied by the requesting FTP client.
password	Pointer to the expected password. The user application callback routine supplies the expected password in the 32-byte string array.

Return values

Return value	Description
0	Success
-1	Server has already been started, registration has been ignored
For app_validp	
0	Username is valid; the password supplied will be used for further authentication
-1	Invalid username

FSSequenceNumber

Returns the current sequence number of the data socket.

This is for callback routines that require an indication of how many times the callback routine has been called.

Format

```
int FSSequenceNumber (unsigned long handle);
```

Arguments

Argument	Description
handle	Handle required for other FTP server API routines.

Return values

Return value	Description
0 or greater	Current sequence number
-1	Handle does not represent a valid data connection

FSStartServer

Starts the server.

Once started, the server ignores all setup calls. Therefore, all server setup calls, such as registering user application callback routines, must be made before you call this routine.

Format

```
int FSStartServer (void);
```

Arguments

none

Return values

Return value	Description
0	Success — the server started successfully
-1	Failure — an error prevented the server from starting

FSTimeout

Changes the FTP server inactivity timeout. If there is no activity on the FTP connection, the server closes the connection after a timeout.

You should call this function after `FSInitialize` and before calling `FSStartServer`.

Format

```
int FSTimeout (unsigned long timeout);
```

Arguments

Argument	Description
timeout	Number of seconds for the FTP server inactivity timeout. Range is 30–900 seconds. Default is 900.

Return values

Return value	Description
0	Success — the timeout has been changed
-1	Failure — the timeout has not been changed because the server data structure has not been initialized or because the server has already been started with a default inactivity timeout

FSUserName

Returns a pointer to the `NULL`-terminated string that contains the username of the current session.

Format

```
char *FSUserName (unsigned long handle);
```

Arguments

Argument	Description
handle	Handle required for other FTP server API routines.

Return values

Return value	Description
non-null	Pointer to the username string
`NULL`	Handle does not represent a valid session

FTP Client API

CHAPTER 4

Overview

The FTP client API enables applications to communicate with FTP servers. Using the FTP client API, you can:

- Transfer large amounts of data to and from a NET+Works node to a central server
- Establish a general-purpose FTP connection to a server and then use FTP to read and write files
- Enable a device to scan directories and change the current directory.

You can use FTP across the Internet.

An FTP client is implemented in NET+Works firmware. All the code needed to connect to servers, scan directories, read files, and write files is implemented in the firmware.

FTP file access

FTP supports two methods for accessing files:

- A file can be read, in which case the read operation starts from the beginning of the file and continues until either the entire file is read or the client disconnects.
- A file can be created and written to. If the specified file already exists on the server, that file is deleted — that is, replaced by the newly created file. The file is written from the beginning; data is appended to the file until the client disconnects (or until the write operation exceeds the memory available).

FTP does not support random access to files.

Include file

Using the FTP client API requires the following header file:

```
fclntapi.h
```

Summary of FTP client API functions

Function	Description
`FCConnect`	Creates a new FTP connection.
`FCDeleteFile`	Deletes a file on the FTP server.
`FCDisconnect`	Disconnects an FTP session.
`FCGetCurrentDir`	Gets the current working directory on the FTP server.
`FCGetData`	Collects data sent by a remote host immediately after a call to `FCRetrieveFile`.
`FCHandleToSocket`	Enables the FTP client application to retrieve the control socket.
`FCListDir`	Gets the file content of the directory specified by a pathname.
`FCMakeDir`	Creates a directory on the FTP server.

Function	Description
FCPutData	Sends data to a remote host immediately after a call to FCStoreFile.
FCRemoveDir	Removes a directory on the FTP server.
FCRetrieveFile	Initiates an FTP GET command.
FCSetCurrentDir	Sets the current working directory on the FTP server.
FCStoreFIle	Initiates an FTP PUT command.

Sample applications

Applications for FTP client functions include:

- **Data collection:** Devices that monitor equipment or that collect data can use FTP to write the data to files.
- **Automatic configuration:** Devices can use FTP to automatically configure at power-up. At power-up, a device can open a predefined FTP file and download configuration data or commands from it. The same file can be shared by all devices in a class or each device can have its own configuration file. To reconfigure a device, a user can simply modify its configuration file and then reset the device. The only configuration data that must be stored in the device's persistent memory is the FTP server address and filename.
- **Firmware update:** You can use FTP to update a device that has flash ROM. Configuration software can send commands to the device to upload a new ROM image. The commands can include an FTP server name and filename. The device then cab use the FTP API to upload the new image.
- **Image transfer:** A scanner can use FTP to store a scanned image. The filename can be specified with a keypad or it can be generated with a program.

FTP client API functions

The following pages describe the FTP client API functions.

FCConnect

Creates a new FTP connection.

Format

```
unsigned long FCConnect(char *ipAddress, char *username,
        char *password);
```

Arguments

Argument	Description
ipAddress	Pointer to a NULL-terminated string denoting a server IP address. It should be in dotted format: *nnn.nnn.nnn.nnn* where *nnn* can be any number between 0 and 255.
username	Pointer to a NULL-terminated string denoting a username.
password	Pointer to a NULL-terminated string denoting a password.

Return values

Return value	Description
n	Unique integer that identifies the FTP session
0	Connection failed

FCDeleteFile

Deletes a specified file on the FTP server.

Format

```
int FCDeleteFile (unsigned long handle, char *file);
```

Arguments

Argument	Description
handle	Identifies the session created by `FCConnect`.
file	Pointer to a `NULL`-terminated string indicating the file to be removed on the server.

Return values

Return value	Description
0	Success
-1	Failure or the file does not exist

FCDisconnect

Disconnects an FTP session.

Format

```
void FCDisconnect (unsigned long handle);
```

Arguments

Argument	Description
handle	Identifies the session created by `FCConnect`.

Return values

Always 0 (zero).

FCGetCurrentDir

Gets the current working directory on the FTP server.

Format

```
int FCGetCurrentDir(unsigned long handle, char *current_dir,
        int maxlen);
```

Arguments

Argument	Description
handle	Identifies the session created by FCConnect.
current_dir	Pointer to an application buffer used to store the current working directory as a NULL-terminated string.
maxlen	Maximum size of the buffer *current_dir* including a NULL terminator.

Return values

Return value	Description
0	Success
-1	Failure or the current directory's length is greater than *maxlen*

FCGetData

Collects data sent by a remote host immediately after a call to `FCRetrieveFile`.

You can call `FCGetData` repetitively to retrieve the data until the function returns an end-of-file.

Format

```
int FCGetData (unsigned long handle, char* buffer,
        int buflen);
```

Arguments

Argument	Description
handle	Identifies the session created by `FCConnect`.
buffer	Pointer to the buffer allocated by the caller to hold data.
buflen	Maximum number of bytes sent for each time `FCGetData` is called.

Return values

Return value	Description
0 or greater	Number of bytes actually read
-1	Failure
-2	No more data — end-of-file reached

FCHandleToSocket

Enables an FTP client application to retrieve the control socket for an ongoing connection.

The socket returned is non-blocking. You can temporarily set it to blocking by calling `setsockopt`.

Format

```
int FCHandleToSocket (unsigned long handle);
```

Arguments

Argument	Description
handle	Identifies the session created by `FCConnect`.

Return values

Return value	Description
0 or greater	Indicates a valid control socket
-1	Handle does not represent a valid session

FCListDir

Gets the list of files within a specified directory.

Format

```
int FCListDir (unsigned long handle, int verbose,
        char *pathname, char *buffer, int len, int *flag)
```

Arguments

Argument	Description
handle	Identifies the session created by `FCConnect`.
verbose	One of the following: ■ `False` — lists only the filename like the `LS` command ■ `True` — includes additional information for the file and directory, such as the `DIR` command
pathname	Pointer to a `NULL`-terminated string indicating the name of the directory.
buffer	Pointer to the buffer allocated by the caller to store the result from the remote host.
len	Length of the buffer.
flag	Pointer to an I/O parameter that indicates whether the data transfer is completed. Set the flag to 0 (zero) when this function is first called. A return value of 1 means that more data is expected. You then call this function repeatedly, with the flag set to 1, until a value of 2 is returned. A return value of 2 indicates no more data.

Return values

Return value	Description
0 or greater	Number of bytes actually read
-1	Failure or the directory does not exist

FCMakeDir

Creates a directory on the FTP server.

Format

```
int FCMakeDir (unsigned long handle, char *dir);
```

Arguments

Argument	Description
handle	Identifies the session created by `FCConnect`.
dir	Pointer to a `NULL`-terminated string indicating the new directory to be created on the server.

Return values

Return value	Description
0	Success
-1	Failure or the directory already exists

FCPutData

Sends data to a remote host immediately after a call to `FCStoreFile`.

Format

```
int FCPutData (unsigned long handle, char *buffer,
       int buflen, int last);
```

Arguments

Argument	Description
handle	Identifies the session created by `FCConnect`.
buffer	Pointer to the buffer allocated by the caller to hold data.
buflen	Maximum number of bytes intended to write.
last	Identifies whether thisis the final call. Specify one of the following: ■ 1 — if this is the final call ■ 0 — if more calls are expected

Return values

Return value	Description
0 or greater	Number of bytes sent
-1	Failure

FCRemoveDir

Removes (deletes) a directory on the FTP server.

Format

```
int FCRemoveDir (unsigned long handle, char *dir);
```

Arguments

Argument	Description
handle	Identifies the session created by `FCConnect`.
dir	Pointer to a `NULL`-terminated string indicating the directory to be removed on the server.

Return values

Return value	Description
0	Success
-1	Failure or the directory does not exist

FCRetrieveFile

Initiates an FTP GET command to retrieve a specified file.

Once the command is successful, the application must call FCGetData to read the stream data sent by the remote host.

Format

```
int FCRetrieveFile(unsigned long handle, char *pathname,
       int type);
```

Arguments

Argument	Description
handle	Identifies the session created by FCConnect.
pathname	Pointer to a NULL-terminated string indicating the name of file to be retrieved.
type	Identifies the mode of transfer. Specify either: ■ 0 — for ASCII ■ 1 — for binary

Return values

Return value	Description
0	Success
-1	Failure

FCSetCurrentDir

Sets the working directory on the FTP server.

Format

```
int FCSetCurrentDir(unsigned long handle, char *dir);
```

Arguments

Argument	Description
handle	Identifies the session created by FCConnect.
dir	Pointer to a `NULL`-terminated string indicating the new working directory on the server.

Return values

Return value	Description
0	Success
-1	Failure

FCStoreFile

Initiates an FTP PUT command to send a specified file to the server.

Once the command is successful, the application must call `FCPutData` to write the stream data to the remote system.

Format

```
int FCStoreFile(unsigned long handle, char *pathname,
        int type);
```

Arguments

Argument	Description
handle	Identifies the session created by FCConnect.
pathname	Pointer to a `NULL`-terminated string indicating the file on the remote system.
type	Identifies the mode of transfer. Specify one of the following: ■ 0 — for ASCII ■ 1 — for binary

Return values

Return value	Description
0	Success
-1	Failure

DNS API

CHAPTER 5

Overview

The Domain Name Service (DNS) API enables applications to access DNS servers on a network to translate DNS names into IP addresses.

DNS is used to associate names with TCP/IP addresses. Instead of using a number sequence to remember a device, this allows you to assign a name that is more easily associated with the device.

You must register a domain name server before calling `DNSgethostbyname`. There are two ways to do this:

- Calling `DNSAddServer`
- Using DHCP during startup to configure IP parameters

Include file

Using the DNS API requires the following header file:

```
dnsc_api.h
```

Summary of DNS API functions

Function	Description
`DNSAddServer`	Adds a server to the list of known DNS servers.
`DNSRemoveServer`	Removes a server from the list of known DNS servers.
`DNSGetServers`	Gets a current list of known DNS servers.
`DNSgethostbyname`	Gets an IP address for a given domain from list of known DNS servers.

DNS API functions

The following pages describe the DNS API functions.

DNSAddServer

Adds a server to the list of known DNS servers.

Format

```
int DNSAddServer (unsigned long server);
```

Arguments

Argument	Description
server	IP address (in host byte order) of the server to be added.

Usage notes

- When you get IP parameters from DHCP, you need not call this function, since it is called automatically by the DHCP client software.
- Call this function before using any other DNS function.
- Maximum number of servers supported is 3.

Return values

Return value	Description
0	Success
-1	Failure

DNSRemoveServer

Removes a server from the list of known DNS servers.

Format

```
int DNSRemoveServer (unsigned long server);
```

Arguments

Argument	Description
server	IP address (in host byte order) of the server to be added.

Return values

Return value	Description
0	Success
-1	Failure

DNSGetServers

Gets the list of the known server IP addresses currently being used by the DNS software.

Format

```
int DNSGetServers (unsigned long *servers, int size);
```

Arguments

Argument	Description
servers	Pointer to an array to hold the server IP addresses.
size	Length of the *servers* array. Since the maximum number of servers is 3, the array should contain at least three elements.

Return values

Return value	Description
integer	Number of servers returned in the array A value of 0 (zero) indicates that there are no servers in use.
-1	Failure

DNSgethostbyname

Gets the IP address of the given domain name from the list of known DNS servers.

At least one server should be registered prior to using the `DNSgethostbyname` routine. A server is registered by means of a successful `DNSAddServer` call or configured by means of DHCP.

Format

```
unsigned long DNSgethostbyname (char *name);
```

Arguments

Argument	Description
name	Pointer to a `NULL`-terminated string containing the domain name to be resolved.

Return values

Return value	Description
integer	IP address (in host byte order)
0	Failed to resolve the domain name

E-mail Services API

CHAPTER 6

Overview

You can use the e-mail services API to customize a mail client according to your application's needs. The API provides the basic framework to communicate with:

- POP3 (Post Office Protocol V.3) servers

  This includes allowing the application to take control when messages are received from the server.

- SMTP (Simple Mail Transport Protocol) servers to send e-mail

Include file

Using the e-mail services API requires the following header file:

```
mailcapi.h
```

Summary of e-mail services API functions

Function	Description
MCCreate	Creates and initializes a new mail client.
MCReadLogin	Logs in to a mail server using a provided username and password.
MCReadLogout	Logs out of a mail server.
MCStat	Sends a STAT command to a POP3 mail server.
MCRegisterLIST	Registers a function called by the mail API when processing data (see MCRegisterRETR, below).
MCList	Sends a LIST command to a POP3 server.
MC_get_msg_count	Retrieves the number of messages for a logged-in user on an attached mail server.
MCRegisterRETR	Registers a function called by the mail client API when processing data (see MCRetr, below).
MCRegisterRETRWithAttachments	Registers a function called by the mail API when processing data (see MCRetrWithAttachments, below).
MCRetr	Sends the RETR command to a POP3 server.
MCRetrWithAttachments	Retrieves a mail message with up to three attachments.
MCNoop	Sends the NOOP command to a POP3 server to check if you are connected.
MCClose	Closes a mail client.

Function	Description
`MCDele`	Sends the `DELE` command to a POP3 server.
`MCRset`	Sends the `RSET` command to a POP3 server.
`MCSendSimpleMail`	Sends an ASCII-text message to the SMTP server of the mail client.
`MCSendSimpleMailWithDomain`	Sends an ASCII-text message to the SMTP server of the mail client and inserts the domain name into the `HELO` greeting with the server.
`MCSendMailWithAttachments`	Sends a mail message with up to three attachments.
`MCSendMailWithAttachmentsAndDomain`	Sends a mail message with up to three attachments and inserts the domain name into the `HELO` greeting with the server.

Sample applications

Sample applications for the e-mail services API include:

- Sending a message to a mailbox

  POP3 would then be used to read the message.

- Using e-mail to allow two devices to communicate over the Internet

  Since e-mail is relatively reliable, if the destination device is down, the message is held until the device comes back online. If the message cannot be delivered for any reason, the Internet mail system automatically sends a notification message to the sender or administrator.

- Data collection

  For example, some security systems take a picture of persons entering a building or accessing some facility or piece of equipment. The camera could send these pictures via e-mail to a central server for storage.

  A second example is for a network of weather stations that could periodically send mail messages containing current weather conditions at each location. The central server could collect and process the data from the mail messages.

- Alerts where high and low marks in a reading could generate a mail message

Memory usage

Calling the e-mail services API results in the dynamic creation of an object, approximately 2.5 Kb per call.

E-mail services API functions

The following pages describe the e-mail services API functions.

MCCreate

Creates and initializes a new mail client. The new client is initialized by opening a socket connection to the mail server (if specified), and stores the SMTP port and IP address (if specified).

Format

```
unsigned long MCCreate (int type, int sport, char *saddr,
        int smtp_port, char *smtp_addr);
```

Arguments

Argument	Description
type	Type of connection to open for reading incoming mail. Supports POP3 only.
sport	Port of the mail server from which the client will read mail.
saddr	Pointer to the IP address or host name of the mail server from which the client will read mail.
smtp_port	SMTP server port to connect to when sending mail.
smtp_addr	Pointer to the SMTP server IP address to connect to when sending mail.

Return values

Return value	Description
non-zero	Identifies the new mail client This indicates a successful connection to the mail server, a successful storing of the SMTP information, or both.
0	Failure — unable to create the mail client This indicates that no part of the initialization sequence requested could be completed.

MCReadLogin

Logs in to a mail server using the provided username and password.

Format

```
int MCReadLogin(unsigned long handle, char *user,
        char *pass);
```

Arguments

Argument	Description
handle	Unique identifier of the mail client.
user	Pointer to the username for logging in to the server.
pass	Pointer to the password for logging in to the server.

Return values

Return value	Description
0	Success
-1	Failure

MCReadLogout

Logs out of a mail server.

Format

```
int MCReadLogout (unsigned long handle);
```

Arguments

Argument	Description
handle	Unique identifier of the mail client.

Return values

Return value	Description
0	Success
-1	Failure

MCStat

Gets the number of messages on a mail server.

Format

```
int MCStat (unsigned long handle);
```

Arguments

Argument	Description
handle	Unique identifier of the mail client.

Return values

Return value	Description
0	Success
-1	Failure

MCRegisterLIST

Registers a user function called by the e-mail client API to process the data as a result of calling `MCList`.

If there is no user-specified function, `MCList` does nothing with the received data.

Format

```
int MCRegisterLIST (unsigned long handle,
        int (*app_LISTp)(char *buffer, unsigned long size,
        unsigned long unused1, unsigned long unused2));
```

Arguments

Argument	Description
handle	Unique identifier of the mail client.
app_LISTp	Pointer to the user function that will be called by `MCList` to handle the received data.
buffer	Pointer to the data returned from `MCList`.
size	Size of the buffer.
unused1	Not used.
unused2	Not used.

Return values

Return value	Description
0	Success
-1	Failure

MCList

Sends the `LIST` command to a POP3 mail server.

The `LIST` command lists one of the following:

- All messages by a numeric value and a byte count
- An individual message by its numeric value and byte count

Format

```
int MCList(unsigned long handle, int msg, int raw);
```

Arguments

Argument	Description
handle	Unique identifier of the mail client.
msg	Number of the message to list, or 0 to list all messages.
raw	Indicates, when set to a non-zero value, that any data passed back to the user should not be processed in any way. Thus, any special characters or codes will not be removed from the text.

Return values

Return value	Description
0	Success
-1	Failure

MC_get_msg_count

Retrieves the number of messages for a user who is logged in on an attached mail server.

Format

```
int MC_get_msg_count(unsigned long handle);
```

Arguments

Argument	Description
handle	Unique identifier of the mail client.

Return values

Return value	Description
0 or greater	Success — this is the actual message count
-1	Failure

MCRegisterRETR

Registers a user function called by the e-mail client API to process the data as a result of calling `MCRetr`.

If there is no user-specified function, `MCRetr` will do nothing with the received data.

Format

```
int MCRegisterRETR (unsigned long handle, int (*app_RETRp)
        (char *data, unsigned long size,
        unsigned long unused, unsigned long status));
```

Arguments

Argument	Description
handle	Unique identifier of the mail client.
app_RETRp	Pointer to the user function that will be called by `MCRetr` to handle the received data.
data	Data returned from `MCRetr`.
size	Size of the data.
unused	Not used.
status	One of the following: ■ `MC_MORE_TO_COME` — indicates “more to come,” that is, another call to *app_RETRp* will follow ■ `MC_OK` — message complete and successful ■ `MC_MSG_TOO_BIG` — message was delivered but truncated due to a size limitation of the internal client buffers (maximum 8192 bytes)

Return values

Return value	Description
0	Success
-1	Failure

MCRegisterRETRWithAttachments

Registers a user function called by the e-mail client API to process the data as a result of calling `MCRetrWithAttachments`.

If there is no user-specified function, `MCRetrWithAttachments` does nothing with the received data.

Format

```
int MCRegisterRETRWithAttachments (unsigned long handle,
        int (*app_RETRWAp) (char * bodybuf, int s2bodybuf,
        char *buf1, int s2buf1, char *buf2, int s2buf2,
        char *buf3, int s2buf3, int overflow));
```

Arguments

Argument	Description
handle	Unique identifier of the mail client.
app_RETRWAp	Pointer to the user function called by `MCRetrWithAttachments` to handle the received data.
bodybuf	Pointer to the message body.
s2bodybuf	Size of the *bodybuf* buffer.
buf1	Pointer to the first attachment (if any).
s2buf1	Size of the *buf1* buffer.
buf2	Pointer to the second attachment (if any).
s2buf2	Size of the *buf2* buffer.
buf3	Pointer to the third attachment (if any).
s2buf3	Size of the *buf3* buffer.
overflow	Flag set if any of the buffers exceeded the maximum.

Maximum size for any buffer is 8192 bytes.

Return values

Return value	Description
0	Success
-1	Failure

MCRetr

Sends the `RETR` command to a POP3 mail server.

It then passes any data received from a mail server, as a result of this call, to the user-specified function set up by `MCRegisterRETR`.

Format

```
int MCRetr(unsigned long handle, int msg, int raw);
```

Arguments

Argument	Description
handle	Unique identifier of the mail client.
msg	Number of the message to retrieve. Must be a non-zero value.
raw	Indicates, when set to a non-zero value, that any data passed back to the user should not be processed in any way. Thus, any special characters or codes will not be removed from the text.

Return values

Return value	Description
0	Success
-1	Failure

MCRetrWithAttachments

Retrieves a mail message with up to three attachments.

Format

```
int MCRetrWithAttachments (unsigned long handle, int msg
        char *bodybuf, unsigned long bufsize, char *buf1,
        unsigned int len1, char *buf2, unsigned int len2,
        char *buf3, unsigned int len3);
```

Arguments

Argument	Description
handle	Unique identifier of the mail client.
msg	Number of the message to retrieve. Must be a non-zero value.
**bodybuf*	Pointer to the message body.
bufsize	Size of the *bodybuf* buffer.
**buf1*	Pointer to the first attachment (if any).
len1	The size of the *buf1* buffer.
**buf2*	Pointer to the second attachment (if any).
len2	Sizeof the *buf2* buffer.
**buf3*	Pointer to the third attachment (if any).
len3	Size of the *buf3* buffer.

Return values

Return value	Description
0	Success
1	Buffer not big enough
-1	Error

MCNoop

Sends the `NOOP` command to a POP3 mail server to check if you are connected.

Format

```
int MCNoop(unsigned long handle);
```

Arguments

Argument	Description
handle	Unique identifier of the mail client.

Return values

Return value	Description
0	Success
-1	Failure

MCClose

Closes a mail client object.

This results in the following:

- All socket connections being used are closed down.
- All memory allocated during the object creation is freed.

Format

```
int MCClose (unsigned long handle);
```

Arguments

Argument	Description
handle	Unique identifier of the mail client.

Return values

Return value	Description
0	Success
-1	Failure

MCDele

Sends the `DELE` command to a POP3 mail server to delete a message on the server.

Format

```
int MCDele (unsigned long handle, int msg);
```

Arguments

Argument	Description
handle	Unique identifier of the mail client.
msg	Number of the message to delete.

Return values

Return value	Description
0	Success
-1	Failure

MCRset

Sends the RSET command to a POP3 mail server.

This is for a global undelete; it resets the status for a message marked for deletion to "undelete."

Format

```
int MCRset (unsigned long handle);
```

Arguments

Argument	Description
handle	Unique identifier of the mail client.

Return values

Return value	Description
0	Success
-1	Failure

MCSendSimpleMail

Sends an ASCII-text mail message to the SMTP server of the mail client.

Format

```
int MCSendSimpleMail (unsigned long handle, char *from,
        char *to_addr, char *subject, char *buf,
                int buflen)
```

Arguments

Argument	Description
handle	Unique identifier of the mail client.
from	Pointer to the sender's e-mail address. Limit 31 characters.
to_addr	Pointer to the recipient's e-mail address. Limit 31 characters.
subject	Pointer to the subject of the message. Limit 64 characters.
buf	Pointer to the message body. Must be `NULL`-terminated.
buflen	The size of the body buffer. Maximum 8192 bytes.

Return values

Return value	Description
0	Success
2	Mail pending
Failures	
-1	`To`: address missing
-2	Missing address
-3	Aborted — no mail server available
-4	Aborted — TPC error
-5	Message cancelled
-6	No mailbox

Return value	Description
-7	Syntax error
-8	Mailbox busy
-9	General abort
-10	Failure due to unknown error

MCSendSimpleMailWithDomain

Sends an ASCII-text mail message to the SMTP server of the mail client, and also inserts the domain name into the `HELO` greeting with the server.

Format

```
int MCSendSimpleMailWithDomain (unsigned long handle,
        char *from, char *to_addr, char *subject, char *buf,
        int buflen, char *domain);
```

Arguments

Argument	Description
handle	Unique identifier of the mail client.
from	Pointer to the sender's e-mail address. Limit 31 characters.
to_addr	Pointer to the recipient's e-mail address. Limit 31 characters.
subject	Pointer to the subject of the message. Limit 64 characters.
buf	Pointer to the message body. Must be `NULL`-terminated.
buflen	Size of the body buffer. Maximum 8192 bytes.
domain	Pointer to the domain name used in the `HELO` statement.

Return values

Same as for `MCSendSimpleMail`

MCSendMailWithAttachments

Sends a mail message with up to three attachments.

Format

```
int MCSendMailWithAttachments (unsigned long handle,
        char *from, char *to_addr,char *subject,
        char *bodybuf, int buflen, char *buf1, int len1,
        char *buf2, int len2, char *buf3, int len3));
```

Arguments

Argument	Description
handle	Unique identifier of the mail client.
from	Pointer to the sender's e-mail address. Limit 31 characters.
to_addr	Pointer to the recipient's e-mail address. Limit 31 characters.
subject	Pointer to the subject of the message. Limit 64 characters.
bodybuf	Pointer to the message body which must be `NULL`-terminated.
buflen	Size of the body buffer. Maximum 8192 bytes.
buf1	Pointer to the first attachment (if any).
len1	Size of the *buf1* buffer.
buf2	Pointer to the second attachment (if any).
len2	Size of the *buf2* buffer.
buf3	Pointer to the third attachment (if any).
len3	Size of the *buf3* buffer.

Return values

Same as for `MCSendSimpleMail`

MCSendMailWithAttachmentsAndDomain

Sends a mail message with up to three attachments, and inserts the domain name into the `HELO` greeting with the server.

Format

```
int MCSendMailWithAttachmentsAndDomain
       (unsigned long handle, char *from, char *to_addr,
       char *subject, char *bodybuf, int buflen, char *buf1,
       int len1, char *buf2, int len2, char *buf3, int len3,
       char *domain);
```

Arguments

Argument	Description
handle	Unique identifier for the mail client.
from	Pointer to the sender's e-mail address. Limit 31 characters.
to_addr	Pointer to the recipient's e-mail address. Limit 31 characters.
subject	Pointer to the subject of the message. Limit 64 characters.
bodybuf	Pointer to thethe message body; must be `NULL`-terminated.
buflen	Size of the body buffer. Maximum 8192 bytes.
buf1	Pointer to the first attachment (if any).
len1	Size of the *buf1* buffer.
buf2	Pointer to the second attachment (if any).
len2	Size of the *buf2* buffer.
buf3	Pointer to the third attachment (if any).
len3	The size of the *buf3* buffer.
domain	Pointer to the domain name used in the `HELO` statement.

Return values

Same as for `MCSendSimpleMail`

SNMP API

CHAPTER 7

Overview

The Simple Network Management Protocol (SNMP) is for communicating management and diagnostic information between the agents in the network and network management stations.

SNMP agent

NET+OS supports SNMP Version 2, which is defined in:

- RFC 1155 and 1157
- RFC 1212 and 1213
- RFC 1901 – 1906

The SNMP agent provides complete MIB-II support as defined in RFC 1213 (the Internet MIB).

The SNMP agent listens on UDP port 161 for incoming SNMP commands from an SNMP management station.

An SNMP command takes the form of a protocol data unit (PDU).

Commands

The SNMP agent supports the commands in the following table:

Command	Description
`GetRequest/ SetRequest`	Supplies a list of objects and, possibly, values they are to be set to. The agent returns a `GetResponse` which informs the management station of the results of a `GetRequest` or `SetRequest` by returning an error indication and a list of variable / value bindings.
`GetNextRequest`	Performs table tranversal and in other cases where the management station does not know the exact MIB name of the object it wants. `GetNextRequest` does not require an exact name to be specified; if no object exists of the specified name, the next object in the MIB is returned. Note that to support this, MIBs must be strictly ordered sets.
`Trap`	The only PDU sent by an agent on its own initiative. Used to notify the management station of an unusual event that may demand further attention (like a link going down).
`GetBulk`	Bulk variable retrieval.

The SNMP agent runs as a daemon, so when SNMP is loaded, it starts a new task that listens for incoming SNMP packets. The communities, enterprise number, and default trap IP address are defined in the `snmp_api.c` file supplied with the NET+OS board support package (BSP).

The authentication scheme checks the community name, so any station in one of the communities is accepted. The variables are readable by all listed communities. Those specified as being writable in RFC 1213 can be written to only by the private community.

All MIB-II objects described are supported.

Variables

The following variables are supplied through the SNMP API described in this chapter. The default values are provided with the agent, but you can modify these values to reflect their application. Note that some variables cannot be set with SNMP.

sysDescr

A description of the entity, including the full name and version identification of the system's hardware type, software operating system, and networking software.

Cannot be set using SNMP.

Maximum length is 255 characters — only printable ASCII characters.

sysObjectID

The vendor's authoritative identification of the network management subsystem contained in the entity. This value is allocated within the subtree 1.3.6.1.4.1 and provides an easy and unambiguous means for determining what kind of device is being managed.

For example, if vendor "NetSilicon, Inc." was assigned the subtree 1.3.6.1.4.1.901, it could assign the identifier 1.3.6.1.4.1.901.1.1 to its network device.

Cannot be set using SNMP.

sysContact

The identification of the contact person for the managed node, along with information on how to contact this person.

Maximum length 255 characters.

sysName

The administratively assigned name for this managed node — by convention, the node's qualified domain name.

Maximum length 255 characters.

sysLocation

The physical location of this node (for example, “Testing Lab”).

Maximum length 255 characters.

sysServices

A sum indicating the services that this entity primarily offers. Initially 0 (zero).

Cannot be set using SNMP.

For each layer (L) in the range 1–7 for which this node performs transactions, $2^{(L-1)}$ is added to the sum. For example, a node that performs primarily routing functions would have a value of 4 (that is, $2^{(3-1)}$). Note that in the context of the Internet protocols, values should be calculated accordingly:

Layer 1 — Physical (for example, repeaters)

Layer 2 — Datalink / subnetwork (for example, bridges)

Layer 3 — Internet (for example, IP gateways)

Layer 4 — End-to-end (for example, IP hosts)

Layer 7 — Applications (for example, mail relays)

For systems including OSI protocols, layers 5 and 6 can also be counted.

Memory usage

The memory usage for the SNMP API is as follows:

- Stack size = 4096 bytes
- Global data = 8192 bytes

Include file

Using the SNMP API requires the following header file:

- `snmpapi.h`

Using the SNMP extensions requires the following header files:

- `asn1.h`
- `snmpimpl.h`
- `snmp.h`
- `snmpvars.h`
- `man_agnt.h`

Summary of SNMP API functions

Function	Description
`snmpd_load`	Starts SNMP.
`SNMPSendTrap`	Sends a trap message.
SNMP extensions	
`snmpAllocateFieldBuffer`	Allocates a buffer for an octet string field.
`snmpEncodeIndices`	Encodes index of a columnar object .
`snmpExtractField`	Copies a columnar object in a table row into a buffer.
`snmpExtractIndices`	Decodes object index information for a columnar object.
`snmpFreeBufferLater`	Frees a buffer when the SNMP agent finishes processing a request.
`snmpFreeOctetStringBuffers`	Frees any octet string buffers allocated by `snmpReadRow`.
`snmpFreeIndices`	Frees a buffer allocated by `snmpExtractIndices`.
`snmpGetFieldCode`	Returns a *hint code* that indicates which columnar object in a table is being accessed.
`snmpGetVariableIdentifier`	Returns the identifier to look up a MIB object in the management database.

Function	Description
snmpGetVariableInfo	Returns a pointer to the element used to register a MIB object with the management API.
snmpReadObject	Reads a MIB object into a buffer.
snmpReadRow	Reads a row from a table.
snmpSetField	Writes the value of a columnar object into a row buffer.
snmpWriteObject	Writes the value of a MIB object into the management database.
Stub functions for setting and getting SNMP variables	
SNMPGetSysContact	Returns a pointer to the sysContact variable.
SNMPGetSysDescr	Returns a string containing the SysDescr variable.
SNMPGetSysLocation	Returns a pointer to the sysLocation variable.
SNMPGetSysName	Returns a pointer to the sysName variable.
SNMPGetSysObjectID	Returns a pointer to an array containing the sysObjectID.
SNMPGetSysServices	Returns the value of the sysServices variable.
SNMPSetSysContact	Sets the sysContact variable.
SNMPSetSysLocation	Sets the sysLocation variable.
SNMPSetSysName	Sets the sysName variable.

SNMP API functions

The following pages describe the SNMP API functions.

snmpd_load

Starts SNMP. The task name is `tSNMP`.

Format

```
int snmpd_load (void);
```

Arguments

none

Return values

Return value	Description
0	Agent started successfully
-1	Error or timeout occurred, agent did not start

SNMPSendTrap

Sends an SNMP trap message to a specified IP address. The trap message is generated by the arguments pass into this function.

Format

```
int SNMPSendTrap (int generic_trap, int specific_trap,
    int num_variables, struct variable_t *variables_ptr,
    int variables_len, char* community [],
    SID_T epriseOID[22], int epriseOIDLen,
    unsigned long ip_address);
```

Arguments

Argument	Description
generic_trap	Type of generic trap to generate.
specific_trap	Enterprise-specific number, used only if generic trap set to (6).
num_variables	Number of variables.
variables_ptr	Pointer to a *variable_t array* (described below).
variables_len	Length of variables in bytes.
community	Community this trap is for.
epriseOID	Pointer to OID of enterprise.
epriseOIDLen	Number of OID digits in enterprise.
ip_address	IP address to send this trap to.

variable_ptr structure definition

The *variables_ptr* has the following format:

```
/* Structure to pass enterprise-specific variables defined in */
/* trapd.h */
struct variable_t {
 SID_T oid_value[24];/* Value of OID in SID_T format. */
 unsigned char value[256];/* Value of the variable of specified type and
length. */
 unsigned char oid_length;/* Number of digits in OID */
```

```
 unsigned char value_type;/* BER coded type */
 unsigned char value_length;/* Length of the value in bytes */
};
```

The following variable types are supported:

- ASN_INTEGER
- ASN_NULL
- ASN_BOOLEAN
- ASN_OCTET_STR
- ASN_OBJECT_ID

Usage notes

Trap messages are sent from the SNMP agent running on the NetSilicon board to an SNMP management station. A trap message indicates that some event occurred on the board which the management station should know about. The SNMP agent supports MIB-II only, which has no mechanism for registering trap addresses or generating traps; the SNMP agent itself does not generate traps. Therefore, an API is provided. The API passes messages to the SNMP agent, which periodically checks for these messages and transmits them.

Trap messages are generated asynchronously by the NetSilicon board and sent to a management station's UDP port 162. The API provided allows you to send a trap message to the SNMP agent, which in turn sends it out to the specified IP address. A trap PDU consists of a trap header followed by any number of interesting variables. The trap header consists of an enterprise OID, agent address, trap type, status code, and time stamp. Each variable consists of a variable name and value.

Return values

Return value	Description
0	Trap was passed successfully
-1	Error occurred, such as the number of variables was greater than `VARIABLES_MAX` defined in `trapd.h`

snmpAllocateFieldBuffer

Allocates a buffer for an octet string field.

The function checks the size of the current octet string buffer. If it is too small to hold the data indicated by the *actionCode* and *info* parameters, that buffer is released and a new one is allocated. Any data contained in the old buffer is copied into the new one.

Format

```
int snmpAllocateFieldBuffer (int actionCode,
        struct varBind *info, void *row);
```

Arguments

Argument	Description
actionCode	Action code passed by the SNMP agent.
info	Structure passed by the agent.
row	Pointer to row buffer to write field into.

Return values

Return value	Description
SNMP_ERR_NOERROR	Success
SNMP_ERR_RESOURCEUNAVAILABLE	Cannot allocate a new buffer

snmpEncodeIndices

Encodes index of a columnar object .

When a columnar object is read using a `GetNext` operation, the index of the object returned is not necessarily the same as the index originally specified by the SNMP console. The read function must encode the index of the object being returned onto the end of the OID that the agent passed to it. The `snmpEncodeIndices` does this encoding, given the indices in an `snmpIndexType` structure.

Format

```
int snmpEncodeIndices (struct variable *vp, oid *name,
        snmpIndexType *index);
```

Arguments

Argument	Description
vp	Pointer to variable agent information about the variable. The SNMP agent passes this as a parameter to read functions, and it is a field in the `varBind` structure passed to write functions.
name	Pointer to the object identifier. The SNMP agent passes this as a parameter to the read function and it is a field in the `varBind` structure passed to write functions.
index	Pointer to index information to be encoded into an OID.

Return values

Return value	Description
0	Index is valid
otherwise	Length of the new OID with the index appended to it

snmpExtractField

Copies a columnar object in a table row into a buffer. The field is copied into the correct format for the columnar object as required by the SNMP agent.

Format

```
void *snmpExtractField (manVarType manInfo, void row,
        int *varLen, void *buffer);
```

Arguments

Argument	Description
manInfo	Pointer to management information about the variable. For the definition of `manVarType`, see the section called "*Defining lists of variables*," in Chapter 15.
row	Pointer to buffer that contains a row.
varLen	Pointer to an integer for the length of the data.
buffer	Pointer to a buffer to store result. If this is `NULL`, the function uses `malloc` to allocate a new buffer that is large enough to hold the data. This buffer must be freed (with `free` or `snmpFreeBufferLater`). If *buffer* is not `NULL`, you must make sure the buffer is large enough to hold the data.

Return values

Return value	Description
`NULL`	Cannot allocate memory for the data, or a field is invalid
otherwise	Pointer to a buffer that contains the data in a form that can be returned to the SNMP agent

snmpExtractIndices

Decodes object index information for a columnar object.

When a columnar object is accessed, the SNMP agent passes the index for the object at the end of the OID in the standard format for SNMP indexes. If the index is simple, it may be easy to decode it directly in the action function. Otherwise, you can use `snmpExtractIndices` to decode the index information and put it inside an `snmpIndexType` structure.

This function dynamically allocates a buffer for the information; this buffer must be freed by calling `snmpFreeIndices`.

Format

```
snmpIndexType *snmpExtractIndices (struct variable *vp,
       oid *name,int length,int isRead, int indexStart);
```

Arguments

Argument	Description
vp	Pointer to agent information about the variable. The SNMP agent passes this as a parameter to read functions, and it is a field in the `varBind` structure passed to write functions.
name	Pointer to the OID. The SNMP agent passes this as a parameter to the read function and it is a field in the `varBind` structure passed to write functions.
length	Length of the OID. The SNMP agent passes this as a parameter to the read function. It is a field in the `varBind` structure passed to write function.
isRead	Should be set if the indices are being extracted for a `Get`or `GetNext` operation.
indexStart	Should be set to indicate the start of the index value in *name*.

Return values

Return value	Description
`NULL`	The MIB object is not a columnar object in a table, or the index is invalid
`otherwise`	Pointer to an `snmpIndexType` structure that contains the index information

snmpFreeBufferLater

Frees a buffer when the SNMP agent finishes processing a request.

When data is returned to the agent by a read function, the buffer that holds the data must be persistent until the SNMP agent finishes processing the current request. Read functions must return pointers to statically allocated buffers or to dynamically allocated buffers that will not be freed until after the request has been completely processed. If a buffer is allocated by `malloc`, the read function can use `snmpFreeBufferLater` to add the buffer onto a list of buffers that the agent frees when it finishes processing the current request.

Format

```
int snmpFreeBufferLater (void *buffer);
```

Arguments

Argument	Description
buffer	Pointer to the buffer to free.

Return values

Return value	Description
SNMP_ERR_NOERROR	Success
SNMP_ERR_RESOURCEUNAVAILABLE	Buffer list is full The action routine should free the buffer immediately, set the length of the data value to zero, and return a NULL pointer.

snmpFreeOctetStringBuffers

Frees any octet string buffers allocated by `snmpReadRow`. No effect if the row does not contain octet strings.

Format

```
void snmpFreeOctetStringBuffers (manVarType *manInfo,
        void *row);
```

Arguments

Argument	Description
manInfo	Pointer to management information about the variable. For the definition of `manVarType`, see the section called "*Defining lists of variables*," in Chapter 15.
row	Pointer to the row buffer.

Return values

none

snmpFreeIndices

Frees a buffer allocated by `snmpExtractIndices`.

Format

```
void snmpFreeIndices (snmpINdenxTuype *index);
```

Arguments

Argument	Description
index	Pointer to index information to be encoded into an OID.

Return values

none

snmpGetFieldCode

Returns an integer value (hint code) that indicates which columnar object in a table is being accessed.

MIBMAN creates hint codes in the MIB header file for all columnar objects in the MIB tables. The hint codes can be used in switch statements to determine which columnar object in a table is being accessed.

Format

```
int snmpGetFieldCode (struct variable *vp);
```

Arguments

Argument	Description
vp	Pointer to agent information about the variable. The SNMP agent passes this as a parameter to read functions, and it is a field in the `varBind` structure passed to write functions.

Return values

Return value	Description
-1	MIB object is not a columnar object in a table
otherwise	Hint code for the columnar object

snmpGetVariableIdentifier

Returns the identifier to look up a MIB object in the management database. This should be treated as read-only.

Format

```
manIDType snmpGetVariableIdentifier (struct variable *vp);
```

Arguments

Argument	Description
vp	Pointer to agent information about the variable. The SNMP agent passes this as a parameter to read functions, and it is a field in the `varBind` structure passed to write functions.

Return values

Return value	Description
`NULL`	MIB object not registered with the management API
otherwise	Identifier for the MIB object

snmpGetVariableInfo

Returns a pointer to the element used to register a MIB object with the management API. This structure should be treated as read-only.

Format

```
manVarType *snmpGetVariableInfo (struct variable *vp);
```

Arguments

Argument	Description
vp	Pointer to agent information about the variable. The SNMP agent passes this as a parameter to read functions, and it is a field in the `varBind` structure passed to write functions.

Return values

Return value	Description
`NULL`	MIB object not registered with the managment API
otherwise	Pointer to the element used to register this object with the management API For the definition of `manVarType`, see the section called "*Defining lists of variables,*" in Chapter 15.

snmpReadObject

Reads a MIB object into a buffer. The data is put into the form expected by the SNMP agent.

Format

```
void *snmpReadObject (struct variable *vp, int *varLen,
       void *buffer)
```

Arguments

Argument	Description
vp	Pointer to agent information about the variable. The SNMP agent passes this as a parameter to read functions, and it is a field in the `varBind` structure passed to write functions.
varLen	Pointer to the length of the result. Set by the caller.
buffer	Pointer to a buffer to store result. If this is `NULL`, the function uses `malloc` to allocate a new buffer that is large enough to hold the data. This buffer must be freed (with `free` or `snmpFreeBufferLater`). If *buffer* is not `NULL`, you must make sure the buffer is large enough to hold the data.

Return values

Return value	Description
`NULL`	Cannot read object or allocate memory for the data
otherwise	Pointer to the buffer containting the data

snmpReadRow

Reads a row from a table.

This function differs from `manGetSnmpRow` in that if the row contains octet strings, `snmpReadRow` automatically allocates the required buffers for them, if necessary.

Note that the indexing scheme used by a table in the management database is not necessarily the same as that used by the corresponding MIB table. It is up to the action routine to translate the SNMP index information into a form that can be used by the management API.

Format

```
MAN_error_type snmpReadRow (manVarType manInfo,
        manTableIndexType *manIndex, void *row);
```

Arguments

Argument	Description
manInfo	Pointer to management information about the variable. For the definition of `manVarType`, see the section called "*Defining lists of variables*," in Chapter 15.
manIndex	Pointer to index information for the row to be read.
row	Pointer to the row buffer.

Return values

Return value	Description
MAN_SUCCESS	Success
MAN_UKNOWN_ID	No variable with the specified identifier
MAN_SEMAPHORE_LOCKED	Timeout expired before variable became ready
MAN_COMMUNICATIONS_FAILURE	Variable is on an external deviced that cannot be communicated with
MAN_INVALID_SUBSCRIPT	No row matching the specified index
MAN_NULL_POINTER	Some parameters are NULL
MAN_OUT_OF_MEMORY	Cannot allocate octet string buffers

snmpSetField

Writes the value of a columnar object into a row buffer.

Format

```
int snmpSetField (int actionCode, struct varBind *info,
       void *row);
```

Arguments

Argument	Description
actionCode	Action code passed by the SNMP agent to determine the value to be written: ■ `SNMP_SET_COMMIT` — write the value in *info->set* ■ `SNMP_SET_UNDO` — write the value in *info->val*
info	Structure passed by the agent.
row	Pointer to row buffer to write field into.

Return values

Return value	Description
SNMP_ERR_NOERROR	Sucess
SNMP_ERR_WRONGLENGTH	Buffer is too short to accept the value
SNMP_ERR_GENERR	Some other error occurred

snmpWriteObject

Writes the value of a MIB object into the management database.

Format

```
int snmpWriteObject (struct variable *vp, union value *value,
        int varLen);
```

Arguments

Argument	Description
vp	Pointer to agent information about the variable. The SNMP agent passes this as a parameter to read functions, and it is a field in the `varBind` structure passed to write functions.
value	Pointer to the value to write.
varLen	Pointer to the length of the result.

Return values

Return value	Description
`SNMP_ERR_NOERROR`	Sucess
`SNMP_ERR_NOSUCHNAME`	Identifier not recognized
`SNMP_BADVALUE`	Value is out of range
`SNMP_ERR_RESOURCEUNAVAILABLE`	Cannot allocate memory
`SNMP_ERR_ GENERR`	Some other error occurred

Stub functions for setting and getting SNMP variables

The SNMP stub functions are implemented in the `snmp_api.c` file supplied with the NET+OS board support package (BSP). You can modify the file according to the needs of your application. The makefile is also supplied to build the SNMP library and link this file.

SNMPGetSysContact

Returns a `NULL`-terminated string pointing to the `sysContact` variable.

Format

```
char *SNMPGetSysContact (void);
```

Arguments

none

Return values

Return value	Description
0	No system contact is available
string	Pointer to the system contact

SNMPGetSysDescr

Returns a `NULL`-terminated string containing the `SysDescr` variable.

Format

```
char *SNMPGetSysDescr (void);
```

Arguments

none

Return values

Return value	Description
0	No system description is available
string	Pointer to the system description

SNMPGetSysLocation

Returns a `NULL`-terminated string pointing to the `sysLocation` variable.

Format

```
char *SNMPGetSysLocation (void);
```

Arguments

none

Return values

Return value	Description
0	No system location is available
string	Pointer to the system location

SNMPGetSysName

Returns a NULL-terminated string pointing to the sysName variable.

Format

```
char *SNMPGetSysName (void);
```

Arguments

none

Return values

Return value	Description
0	No system name is available
string	Pointer to the system name

SNMPGetSysObjectID

Returns a pointer to an array containing the `sysObjectID` variable.

Format

```
unsigned long *SNMPGetSysObjectID (int *size)
```

Arguments

Argument	Description
size	Pointer to an integer set to the number of elements (unsigned longs) in the array returned.

Return values

Return value	Description
0	No `sysObjectID` is available
integer	Pointer to the `sysObjectID` array

SNMPGetSysServices

Returns the value of the `sysServices` variable.

Format

```
char* SNMPGetSysServices (void);
```

Arguments

none

Return values

Return value	Description
0	No system services are available
integer	Value of `sysServices`

SNMPSetSysContact

Sets the SysContact variable.

Format

```
char *SNMPSetSysContact (char *sys_contact, int size);
```

Arguments

Argument	Description
sys_contact	Pointer to a NULL- terminated string containing the definition of SysContact.
size	Number of octets in the string. Not currently used.

Return values

Return value	Description
0	NoSysContact is available
string	Pointer to SysContact

SNMPSetSysLocation

Sets the SysLocation variable.

Format

```
char *SNMPSetSysLocation (char *sys_location, int size);
```

Arguments

Argument	Description
sys_location	Pointer to the NULL-terminated string containing the definition of SysLocation.
size	Number of octets in the string. Not currently used.

Return values

Return value	Description
0	No location is available
string	Pointer to the location

SNMPSetSysName

Sets the `sysName` variable.

Format

```
char *SNMPSetSysName (char *sys_name, int size);
```

Arguments

Argument	Description
sys_name	Pointer to the `NULL`-terminated string containing the definition of `SysName`.
size	Number of octets in the string. Not currently used.

Return values

Return value	Description
0	No system name is available
string	Pointer to the name

Interfacing with SNMP

MIB registration is how your MIB data and code are recognized by the SNMP core agent. MIB registration is required for the agent to be able to recognize how to process an incoming object identifier (OID). MIB groups can be transient in the system. However, they typically register at system initialization and unregister at system termination.

There are two primary data structures associated with MIB definitions in the SNMP agent — the *subtree* structure and the *variable* structure.

Subtrees and variable structures

A *subtree* points to an array of variable structures and is built as a linked list by the agent. Your code will never have to deal with a subtree, but understanding how they are used may be helpful.

A *variable structure* defines an instance of a variable in a MIB definition. To save memory, there are alternative variable structures defined with differing sizes for the final element in the structure, the name or OID element. The maximum size of an OID is 128. Each sub-identifier is a `LONG`. If each variable were defined with its entire OID, the size would be enormous. Therefore, the there are alternative definitions such as *`variable2`* that have only two sub-OIDs in the definition and *`variable3`* with has three sub-OIDs, and so on.

The elements of the variable structure are as follows:

Element	Description
magic	Unique number identifying the variable within the group. Typically the variable number in the OID.
type	One of the following SNMP data types for the variable: ■ `STRING` ■ `BITSTRING` ■ `OPAQUE` ■ `NSAP` ■ `INTEGER` ■ `UINTEGER` ■ `IPADDRESS` ■ `COUNTER` ■ `GAUGE` ■ `TIMETICKS` ■ `OBJID` ■ `COUNTER64` ■ `NULLOBJ`
acl	Read, write, notify access field.
findVar	Pointer to the access routine.
namelen	Number of sub-OIDs in the *name* field.
name	OID of the variable (following the OID for the group).

Elements of a variable structure

An OID can be thought of as containing three parts:

- Group identifier — defined in the subtreet structure
- Variable identifier — typically a single sub-OID
- Index — always 0 for non-table variables

The length of the longest variable identifier is what determines which variable structure to use.

Each subtree must have a unique identifier. This means a MIB group cannot be split into two subtrees. A subtree cannot lexically belong in the middle of another subtree.

MIB registration

MIB registration is done by calling `mibGroupRegister` with the group identifier and variable array for the group. The `mibGroupRegister` function:

- Assures that the group identifier is unique.
- Builds a subtree for the group.
- Inserts the subtree into a linked list of subtrees.

Each subtree entry contains a pointer to a MIB variable array and the group name (OID) for that variable array. Space for the name is allocated in `mibGroupRegister`, so the first parameter may point to dynamic memory.

For example, to register a variable array called `sysGroup`, use the following call:

```
mibGroupRegister (sysOid, sysOidLen,
    (struct variable *)&sysGroup[0],
    SNMP_NENT(sysGroup), sizeof(sysGroup[0]));
```

MIBs within the system are dynamic and therefore transient, depending on whether or not it is currently registered. To unregister a MIB, call `mibGroupUnregister` with the group identifier to be removed.

For example, to remove the `sysGroup` array, use the following call:

```
mibGroupUnregister (sysOid, sysOidLen);
```

After this call, the variables defined in the `sysGroup` array are no longer accessible through SNMP commands. The group could be registered again or replaced by another group with same group identifier.

Access routines

The fourth element of the variable structure is a pointer to the `findVar` routine for the element:

```
void * findVar (struct variable *vp, oid *name,
    int *length, int wantExact, int *varLength,
    setMethod *writeFn);
```

The following table describes the fields in `findVar`:

Field	Description
vp	Pointer to the variable structure entry corresponding to the requested name (OID).
name	The full OID of the variable, and if this is a `GetNext` operation, space for 128 sub-OIDs.
length	Number of sub-OIDs in *name*.
wantExact	If this is non-zero, the variable instance to be returned must match the name exactly. If this is 0, the variable to be returned must be the next lexical OID.
varLength	Pointer to the length of the variable returned.
writeFun	Pointer to the function to call to set this variable. The prototype for the *writeFun* is defined by the `setMethod` type definition (see *"Set routines"*, below

Accessing a scalar variable

In accessing a scalar variable, *wantExact* can be ignored. The calling routine will have already gone to the next OID for a `GetNext` command.

The *vp* parameter points to the variable structure entry for the variable to be returned. The variable structure has a magic number element to indicate which member in the group is wanted. Simply switch on the magic number and set the information for that variable. If the magic number is not known, return `NULL`.

The *name* and *length* will not change.

The *varLength* must be set to the size of the data being returned, and the return value must be a pointer to the data.

The variable length is the number of bytes pointed to by the return pointer, except in the case of an OBJID data type, in which case it is the number of sub- OIDs.

If the variable is settable, the *writeFn* parameter must point to the function for setting variables of this group. If the variable is not settable, *writeFn* should be set to NULL.

Accessing a table variable

In accessing table variables, keep in mind that there may be more than one instance of each variable. To access the proper instance, the name passed in must be parsed into a table index.

The formulas for converting an index to an OID are as follows:

SNMP Types	Formula
STRING BITSTRING NSAP	Length of the data, followed by the data as an array of longs. For example, the string "az" would come before "abc" because the length of azis 2 and the length of abc is 3. The respective indexes as OIDs would be: "2,'a','z'" "3,'a','b','c'" If the field is an implied index, the *length* is not used and, for example, abc would precede az.
INTEGER UINTEGER COUNTER GAUGE TIMETICKS	The number as an unsigned long.

SNMP Types	Formula
IPADDRESS NETWORKADDRESS	The IPADDRESS type is 4 bytes of an IP address in network order. The conversion is simply a long-to- byte conversion. However, IP addresses are stored internally as unsigned longs, so the conversion has to place the data in host order. In the NETWORKADDRESS type, the first sub-OID can be ignored, and the remaining 4 sub-OIDs converted as an IPADDRESS.
OBJID	Length of the OID, followed by the data as an array of longs. If the field is an implied index, the *length* is not used.
OPAQUE COUNTER64 NULLOBJ	Cannot be used in *index* fields.

In the case of an implied index, the length of the field is not part of the OID and therefore not part of the sorting algorithm. The length is extracted by determining the number of remaining sub-OIDs.

To converting an OID to an index, you must know how the SNMP table is indexed to know how to interpret the OID requested. So, for retrieving an instance of data for a table element, the process is:

1. Convert the name parameter to an index.
2. Retrieve the data for that table instance.
3. Switch on the magic number.

and so on, as in the processing of accessing a scalar variable. f the name does not translate to a proper index, or the magic number is bad or the instance for the requested index does not exist, then return NULL.

Accessing the next instance of a table variable

Performing a table `GetNext` is more complex. Keep in mind that that SNMP retrieves table data in column order. In other words,it gets the first element of each table row before getting the second element of the first row, and so on.

If the table you are retrieving data from is sorted in the same order as SNMP indexing, then determining the next entry is a little more straightforward, but possibly still more complex than simply finding the index entry and getting the entry that follows it.

For example, you may have a table with five entries, indexed by vowels in the alphabetical order —

```
{a, e, i, o, u}
```

In such cases, a `GetNext` with an index of `d` returns data for row `e`.

The more complex case is when the table is not in sorted in the same order as SNMP indexing. For example, you may have a table with five entries, indexed by vowels in the order

```
{a, o, u, i, e}
```

If we get the same request — `GetNext` with an index of `d` — the logic in pseudo-code would be:

```
GetNext(after)
best = max
foreach entry in the table
    is entry index > after
        yes, is entry index < best
            yes, best = entry index
next table entry
is best = max
    yes, return entry not found
    no, return entry for best
```

Therefore, to retrieve the next instance of data for a table element, the process is as follows:

1. Convert the name parameter to an index
2. Retrieve the next lexical table instance.
3. Switch on the magic number etc. as in the processing of retrieving a scalar variable.
4. Convert the index to an OID and set the OID length to the number of sub-OIDs for the new name.

If the name does not translate to a proper index, or the magic number is bad, or the requested index is the last entry in the table (or beyond the last entry in the table), return `NULL`.

When a `GetNext` request returns `NULL`, the agent increments to the next variable in the MIB (possibly the next column of the table), clears the index, and requests a `GetNext`is for that variable.

Set routines

`Set` routines all have a prototype that matches the `setMethod` type definition:

```
int functionName (int action, struct varBind *var);
```

The following table describes the fields in the type definition:

Field	Description
functionName	Name of the routine.
action	One of the following: ■ `SNMP_RESERVE` ■ `SNMP_COMMIT` ■ `SNMP_ACTION` ■ `SNMP_FREE` ■ `SNMP_UNDO`
var	Pointer to a variable binding structure.

The variable binding structures comprise the following data:

Element	Description
next	Pointer to the next variable in the Set command. Ignored in set routines.
oid	Pointer to the OID or name of the variable. Ignored in set routines.
oidLen	Length of the OID. Ignored in set routines
setTo	The value that the command is trying to set the variable to. This is a union of different variable types.
setToLen	The size of the *setTo* value in bytes; for OIDs, the number of sub-OIDs.
state	Bit values that describe *setTo* value and the *val* .
type	ASN type of the variable.
val	Value that the variable had before any set action. This is a union of different variable types.
valLen	Size of *val* in bytes; for OIDs, the number of sub-OIDs.
vp	Pointer to the MIB variable table entry. Ignored in set routines.
set	Pointer to the set routine.

The *setTo* and *val* fields are a *value* union. The *value* union comprises the following:

- Unsigned long
- Unsigned long pointer
- Char pointer (string pointer)
- OID pointer
- Counter64 pointer

How to interpret these fields depends on the data type to be set and the flags in the *state* field. All numeric fields — byte, integer, short, long, signed or unsigned — are passed as unsigned longs and can be cast to the proper data type. Use the VAR_SET_VAL bit of the *state* field to determine if the unsigned long value is stored in *setTo* or pointed to by *setTo*. If the bit is set, *setTo* contains the value. The following code fragment shows the typical method for extracting a numeric *setTo* value:

```
int x;
x = (int) ((var->state & VAR_SET_VAL)
    ? var->setTo.intVal : *var->setTo.integer);
```

Use similar code to get the *val* for an UNDO.

For strings, OIDs and counter64 data types, the *setTo* value is always a pointer. For a counter64, it always points to a struct counter64. The length of the buffer pointed to by *setTo* is stored in *setToLen* and is in bytes except for OIDs where it is the number of sub-OIDs.

Set routines are called from within a loop that cycles through each variable in the set command and the SNMP_RESERVE, SNMP_COMMIT, and SNMP_ACTION actions. The rule for an SNMP Set command is that either all variables in the command are set or none of the variables in the command is set. If an error occurs during the reserve, commit or action phases, the processing will break out of the loop and enter a new loop calling the set routines for each variable with an action of SNMP_UNDO. Each variable set routine is called with an SNMP_FREE action. A state variable should be used to track the state of the set process.

The architecture of the process is to reserve space, check for a valid state or transition to a valid state during the reserve phase. During the commit phase, the value should be validated and checked for dependencies. It is important to flush out any errors in these two phases. An error found before the action phase does not need to be undone when another error occurs if the process is followed.

The action phase is when the data is actually set.

The free phase is the opposite of the reserve phase. During this phase, any temporary memory that was allocated should be freed and any state transitions that took place in the reserve phase should transition out of the set state.

An UNDO action will occur only when an error occurred in the RESERVE, COMMIT or ACTION phases. If the data has not been set, the undo can be ignored. If the data has been set, the value of the variable prior to the set will be in the variable binding *val* member, and the variable must be returned to this value. The set routine will be called again with SNMP_FREE, so there is no need to free resources with an SNMP_UNDO action.

Setting scalar variables

For setting a scalar variable, a set routine normally checks the value range in the commit phase and sets the data in the ACTION phase or UNDO phase, and ignores the RESERVE and FREE actions. However, if the variable is dynamic in size, the processing will be very similar to the processing of an existing table row, as described in the following section.

Setting a variable in a table row

To update a variable in an existing table row, the process determines that the row exists in the RESERVE phase. It also reserves temporary storage space and populates it with data from the existing row.

During the COMMIT phase, the process validates the value in the setTo field of the variable binding and overrides the row in temporary storage with the value.

In the ACTION phase, the row in temporary storage update the actual data storage area. Remember to update your state in this phase to indicate that the data has been set. This is necessary to avoid setting the same row twice and to know if you need to recover for an UNDO.

If an UNDO action is received, only data that has been set needs to be undone. Therefore, check the state. If the data has been set, set it back to its previous value using *val* in the variable binding.

During the FREE phase, free up any reserved resources and clear the state.

Setting a variable in a new row added to the table

Updating an element in a non-existent table row is a way of adding a row to a table.

If adding a table row is not a legal SNMP action, the `set` routine must return an error such as `SNMP_ERR_COMMITFAILED` or `SNMP_ERR_RESOURCEUNAVAILABLE`.

If adding a row is legal, the `set` routine must ensure that the row to be added is complete before adding it to the table.

If your MIB table contains a `RowStatus` variable, you may be able to use the *`createAndWait`* value to build a partial row entry, see RFC 2579. The partial entry may need to be maintained in the SNMP temporary storage space until it is ready to be written to the system table. In cases such as this, the state data would need to be maintained between SNMP commands and not cleared during the `FREE` phase.

Error codes for set routines

The return value from a `set` routine is an `SNMP_ERR` error code defined in `snmp.h` as follows:

Return value	Description
`SNMP_ERR_AUTHORIZATIONERROR`	User is not authorized to update the variable
`SNMP_ERR_COMMITFAILED`	Process external to your set routine failed
`SNMP_ERR_INCONSISTENTNAME`	Value to be set (typically an OID) refers to something that does not exist
`SNMP_ERR_INCONSISTENTVALUE`	Cannot set mutually exclusive values
`SNMP_ERR_NOCREEATION`	Illegal index
`SNMP_ERR_NOERROR`	Sucess
`SNMP_ERR_NOTWRITABLE`	Value is not writable This is normally caught by the agent before calling a `set` routine.

Return value	Description
SNMP_ERR_RESOURCEUNAVAILABLE	Reserve phase fails Typically a memory allocation error or some sort of lock out condition.
SNMP_ERR_UNDOFAILED	Your `set` cannot perform an undo
SNMP_ERR_WRONGLENGTH	String or OID is too long or too short
SNMP_ERR_WRONGVALUE	Value is out of range

PPP API

CHAPTER 8

Overview

The Point-to-Point Protocol (PPP) is a communications protocol that allows computer devices to perform network communications through a serial communications line.

Sample PPP applications

Four sample PPP applications are included in the `src\examples\ppp` directory. These samples include both modem and direct serial applications operating in client and server modes. They demonstrate a broad range of functionality, and most user PPP applications should resemble one of these sample programs. If possible, you should develop programs that model these sample programs.

The four sample programs are:

- Echo client to remote echo server through a remote access server (RAS) over a modem on `COM2`
- FTP server available through a direct serial connection on `COM1`
- HTTP server available through a modem connection on `COM2`
- IP routing between a direct serial connection to a modem connection

Sample 1: Echo client to a remote server through a RAS via a modem on COM2

The `echomod2` sample application demonstrates how a client connects to a remote access server (RAS) via modem, as follows:

1. The application actively connects `COM2` via modem to the RAS at 800 555-1212.
2. After the connection is established, the unit connects to an echo server (7.92.186.28) at NetSilicon.
3. An echo test is performed, which sends arbitrarily sized packets of data to the server. The reply is checked for mismatches.

The following figure shows the schematic for this test scenario:

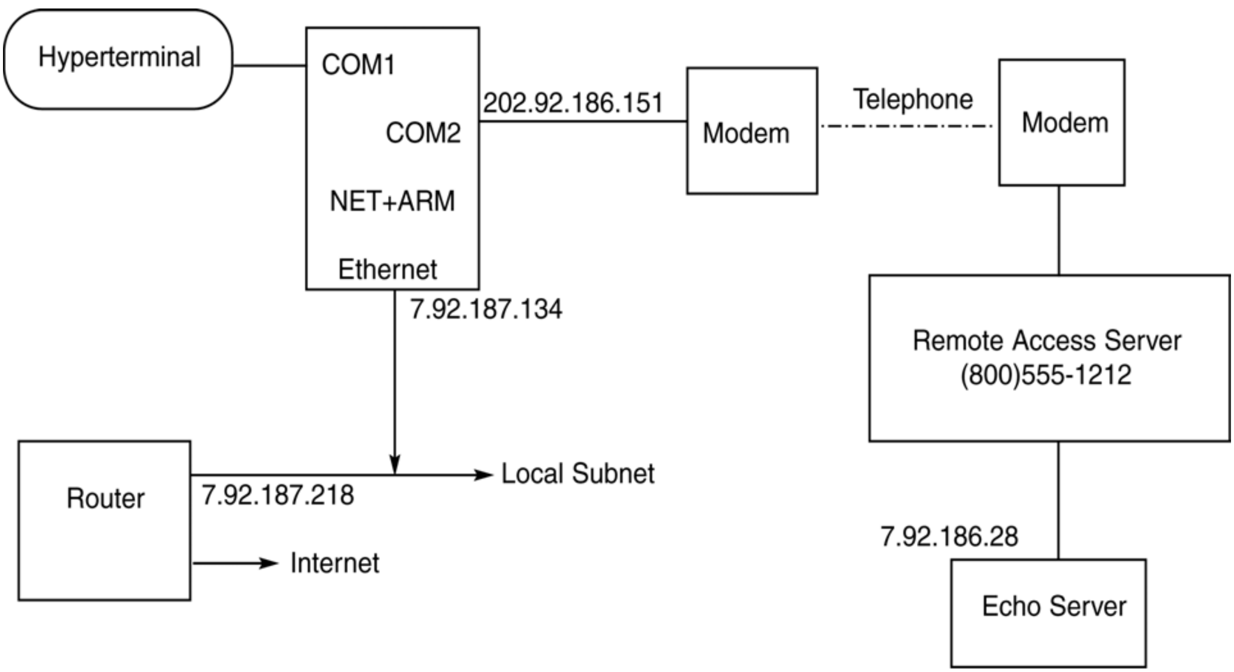


Figure 1: Echomod2 test scenario

Sample 2: FTP server available through direct serial connection on COM1

The `ftpdircom1` sample application demonstrates how a NetSilicon development board can provide services to a directly connected remote client, as follows:

1. The application starts an FTP server on the board.
2. It allows network access to the server via the Ethernet connection (7.92.187.134).
3. It allows access by the remote PPP peer through the serial link (200.92.187.195).

The following figure shows the schematic for this test scenario:

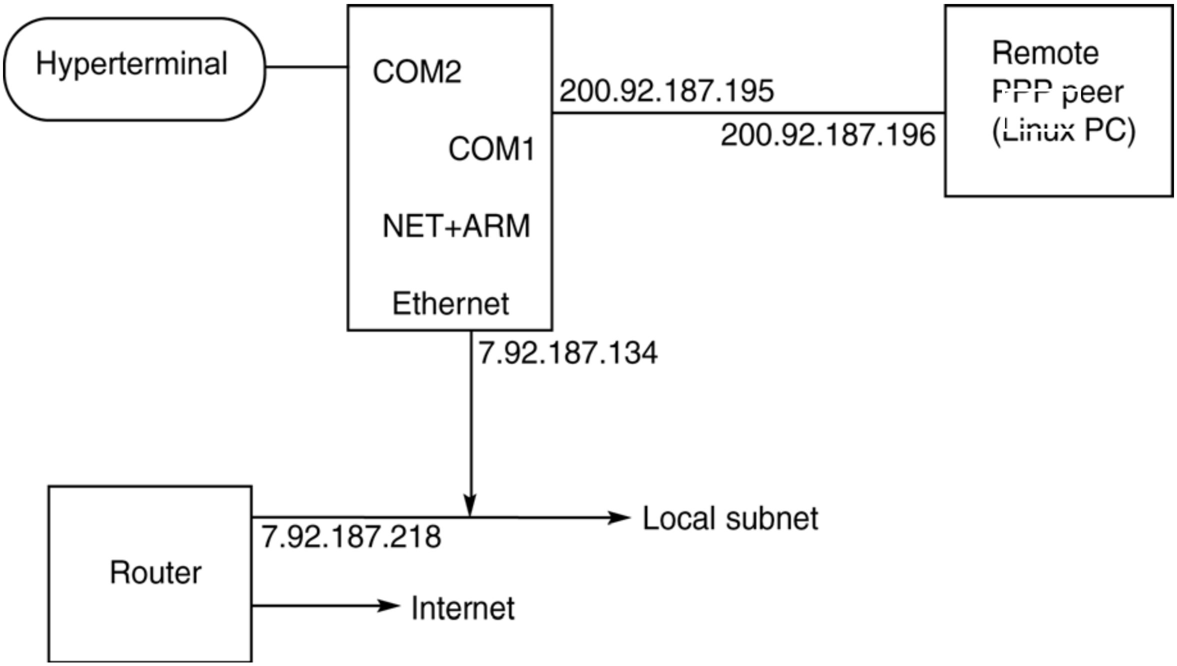


Figure 2: Ftpdircom1 test scenario

Sample 3: HTTP server available through remote access via a modem on COM2

The `httpmod2` sample application demonstrates how a NetSilicon development board can emulate a remote access server and provide services to a client connected through a modem, as follows:

1. The application starts an HTTP server on the unit and passively waits for connections on `COM2` via modem.

   The modem speed is set to 28800 bps.

2. After a connection is established, the operator can use a Web browser on the remote client to browse pages served through the PPP interface by connecting to `http://202.92.186.151/`.

The following figure shows the schematic for this test scenario:

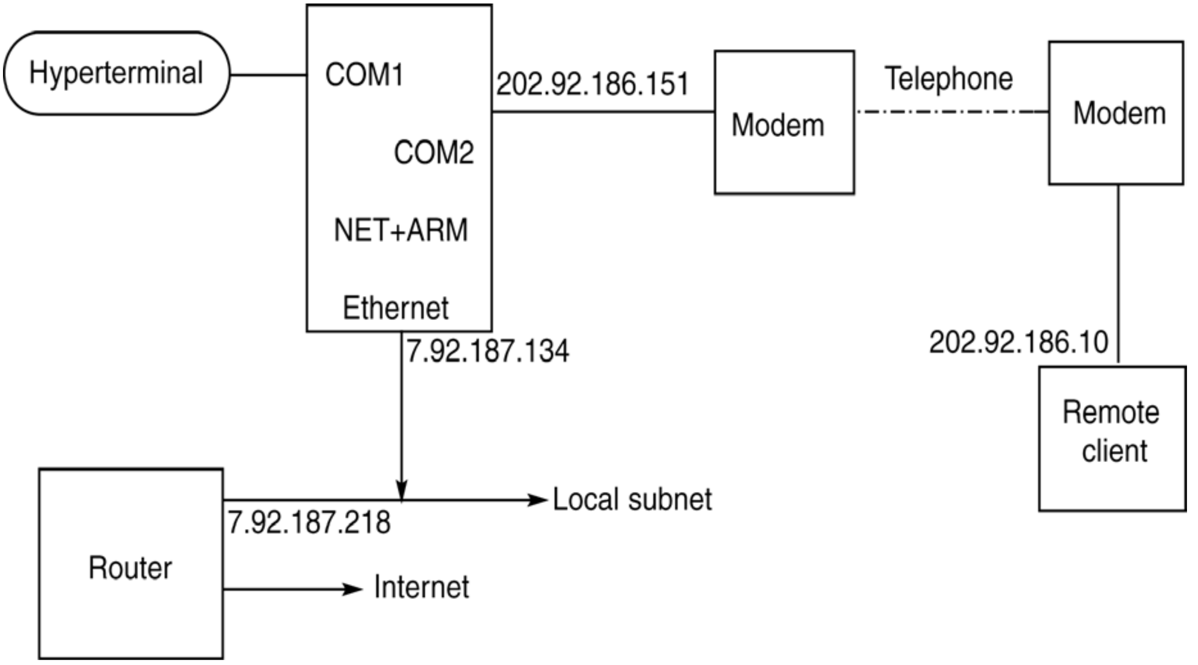


Figure 3: Httpmod2 test scenario

Sample 4: IP routing between a direct serial connection and a modem connection

The `iproute` sample application demonstrates how a NetSilicon development board can connect to two remote devices, as follows:

1. The application uses both ports for PPP, with a direct PPP serial connection on `COM1` running at 57600 bps and a modem connection on `COM2` running at 19200 bps.
2. The application starts an FTP server on the board, which allows network access to the server via an Ethernet connection (7.92.187.134) or a remote PPP peer through either serial link (200.92.187.195 or 202.92.186.151).
3. IP routing can be demonstrated using `PING` in an `xterm` on a Linux system, although entries must be added to the routing tables on the two remote systems. In this scenario:
   - The Linux system routes the `PING` request to the chip.
   - The chip forwards the packet to the NT client via modem.
   - The NT client replies to the chip.
   - The chip forwards the response to the Linux system.

The following figure shows the schematic for this test scenario:

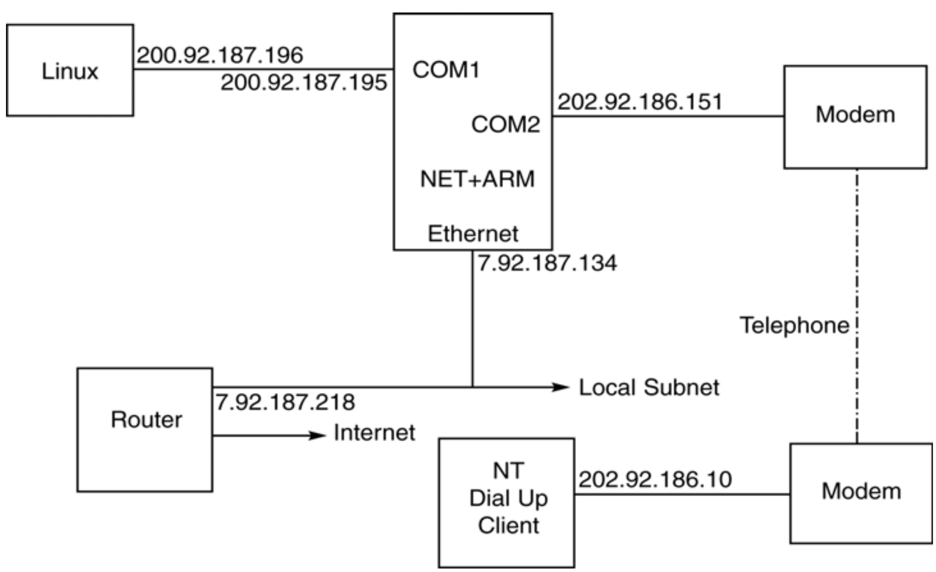


Figure 4: IProute test scenario

Summary of PPP API functions

Function	Description
PPPAddRoute	Adds a routing entry to the IP stack's routing table.
PPPAddUser †	Adds the PAP username / password pair or CHAP ID / secret-key pair of a PPP peer that has access to the system through PAP or CHAP authentication.
PPPCheckLink	Checks the status of PPP link on a specified PPP interface.
PPPCreateDevice	Creates and configures a PPP device on a specified port.
PPPDelRoute	Deletes a routing entry from the IP stack's routing table.
PPPGetPeerAssignedAddress	Gets the peer assigned IP Address of a PPP interface.
PPPModemClose	Closes a specified port and terminates all communications between that port and the modem.
PPPModemInit	Opens and initializes a port to a specified baud rate, and initializes the modem.
PPPSerialClose	Closes a specified port.
PPPSerialInit	Opens and initializes a port to a specified baud rate.
PPPSetAuth	Sets the authentication modes for the specified PPP interface using CHAP or PAP. Also sets the specified PPP interface's PAP username / password and CHAP ID / secret-key pairs.
PPPSetAuthentication †	Sets a specified PPP interface to require or accept CHAP or PAP authentication.

Function	Description
PPPSetModemAutoAnswerRings	Sets the ring count before the modem answers an incoming call.
PPPSetModemDialString	Sets the modem dial string.
PPPSetPeerAddr	Specifies the IP address sent to the client through IPCP. Used when you are using PPP as a server.
PPPSetVJ	Enables or disables Van Jacobson compression or IP header compression.

† = Deprecated or superseded function

State diagrams

PPP API functions must be called in a required sequence. The following figures show the state diagrams of the PPP API for modem and direct serial connections.

The state diagrams contain the following pseudo-states:

- UNINITIALIZED
- STOPPED
- STARTED

For example, you cannot call PPPModemInit before PPPCreateDevice. Calling PPP API functions causes some state transitions.

Functions may work outside of their call state but in such cases, their performance cannot be guaranteed.

The following figure shows the state diagram for a modem connection:

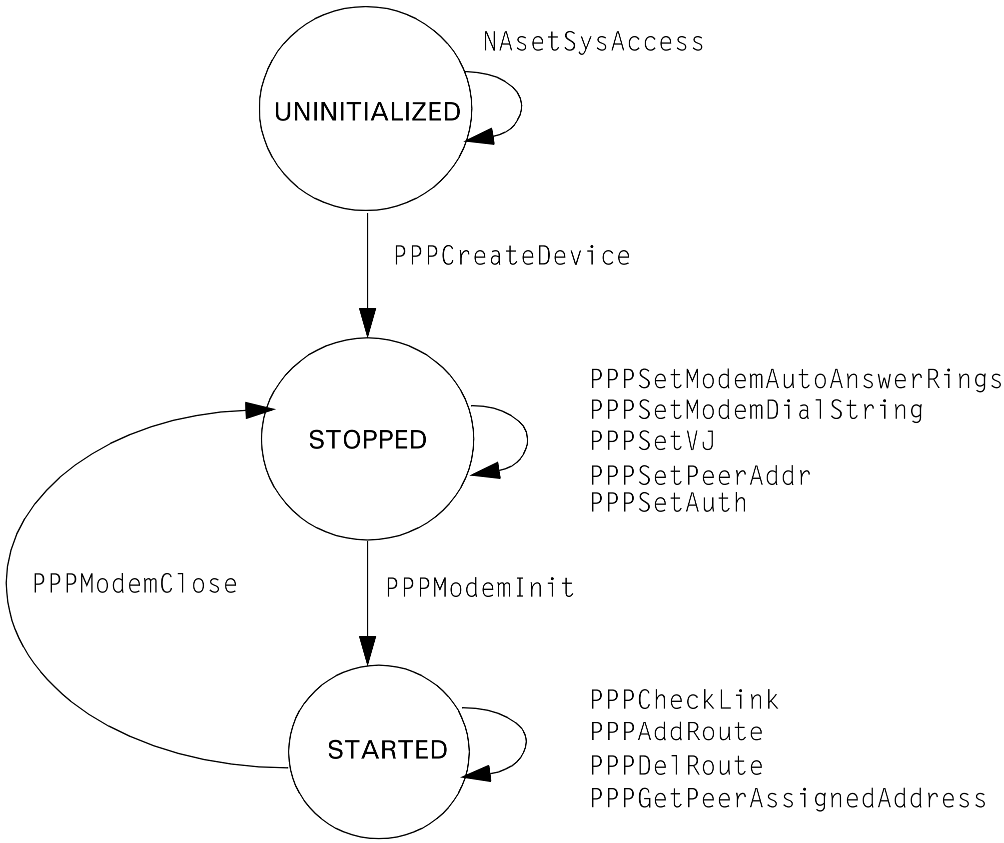


Figure 5: Modem state diagram

The following figure shows the state diagram for direct serial connections:

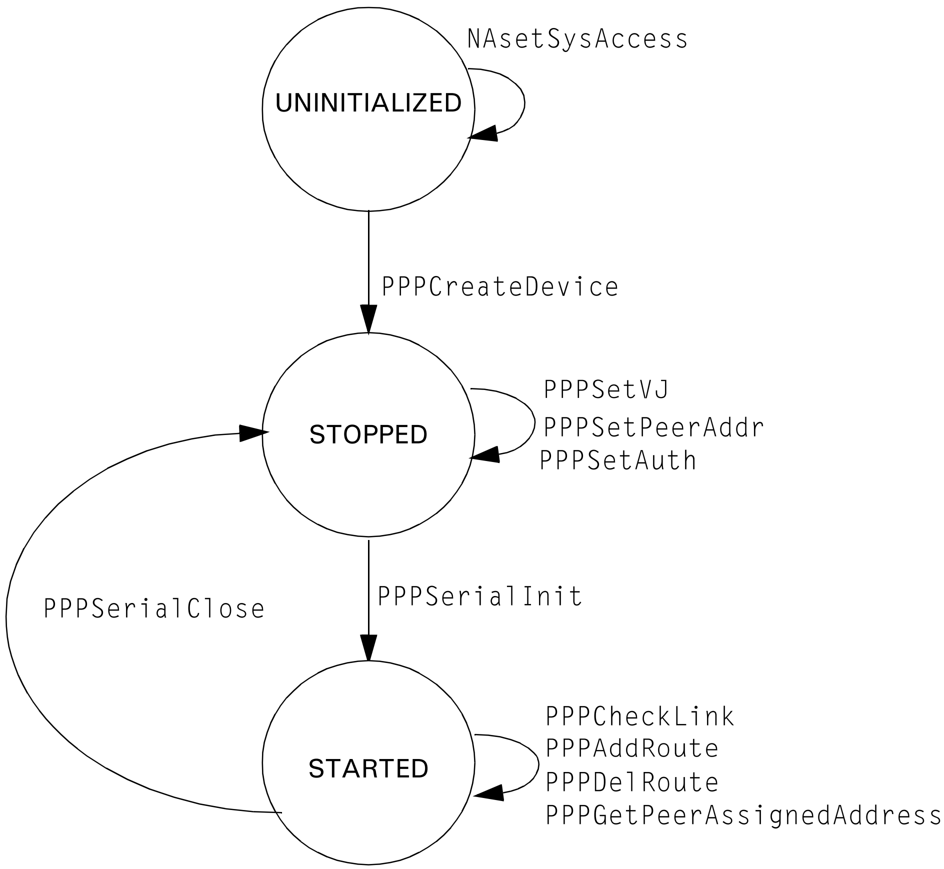


Figure 6: Direct serial state diagram

Include file

Using the PPP API requires the header file:

```
ppp_api.h
```

PPP API functions

The following pages describe the PPP API functions.

PPPAddRoute

Adds a routing entry to the IP stack's routing table.

State

Callable from the STARTED state.

Format

```
int PPPAddRoute (unsigned long destination,
        unsigned long mask, unsigned long gateway,
        int commPort);
```

Arguments

Arguments	Description
destination	IP address of the destination node.
mask	Subnet mask.
gateway	IP address of the gateway node to reach the destination node.
commPort	One of the following: ■ PPP_COMPORT1 ■ PPP_COMPORT2 This argument should match an interface already created by a PPPCreateDevice call.

Return values

Return values	Description
SUCCESS	Route was successfully added to the table
ADD_ROUTE_ERROR	IP stack internal table is full
UNIT_OUT_OF_RANGE	Port has not been created as a PPP port

PPPAddUser

Adds the PAP username / password pair or CHAP ID / secret-key pair of a PPP peer that has access to the system through PAP or CHAP authentication. Also used to update an existing entry in the internal table.

Note: Deprecated function. Instead, the system access database should be populated by calling `NAsetSysAccess`(in `sysAccess.h`).

State

Callable from the `UNINITIALIZED` state.

Format

```
int PPPAddUser (char *username, char *password);
```

Arguments

Argument	Description
username	Pointer to the buffer storing the PAP username or CHAP ID.
password	Pointer to the buffer storing the PAP password or CHAP secret-key.

Return values

Return values	Description
`SUCCESS`	Username / password pair was added to the internal table
`ADD_USR_PWD_ERROR`	Internal table is full or username is null

PPPCheckLink

Checks the status of the PPP link on the specified port.

State

Callable from the `STARTED` state.

Format

```
int PPPCheckLink(unsigned int commPort);
```

Arguments

Arguments	Description
commPort	One of the following: ■ `PPP_COMPORT1` ■ `PPP_COMPORT2` This argument should match an interface already created by a `PPPCreateDevice` call.

Return values

Return values	Description
`SUCCESS`	PPP link is up
`UNIT_OUT_OF_RANGE`	Port has not been created as a PPP port
`PPP_NOT_UP`	PPP link is not up

PPPCreateDevice

Creates and configures a PPP device on the specified port, where the mode argument is used to select either a modem or direct serial connection.

The device IP parameters are configured based on the specified IP address and subnet mask.

State

Callable from the `UNINITIALIZED` state and causes a transition to the `STOPPED` state.

Format

```
int PPPCreateDevice (unsigned int commPort,
        unsigned int mode, unsigned int ipAddress,
        unsigned int subnetMask);
```

Arguments

Arguments	Description
commPort	One of the following: ■ `PPP_COMPORT1` ■ `PPP_COMPORT2` This argument should match an interface already created by a `PPPCreateDevice` call.
mode	Type of serial connection used: ■ `PPP_DIRECT_SERIAL_MODE` ■ `PPP_MODEM_MODE`
ipAddress	IP address (in network byte order) assigned to the PPP device. This IP address must not have a direct connection to any of the network interfaces. (The IP address should not be on any subnet associated with an existing network interface.)
subnetMask	Subnet mask (in network byte order), along with the IP address to define the network address of the device. This network address must be different from all other network interfaces in the unit.

Return values

Return values	Description
SUCCESS	Success
OPEN_PORT_ERROR	Port has already been created by a prior PPPCreateDevice call or the PPP device table is full Only two devices are allowed— one for COM1 and one for COM2.
SYSTEM_RESOURCE_ERROR	Cannot allocate memory

PPPDelRoute

Deletes a routing entry from the IP stack's routing table.

State

Callable from the STARTED state.

Format

```
int PPPDelRoute (unsigned long destination,
        unsigned long mask);
```

Arguments

Arguments	Description
destination	IP address of the destination node.
mask	Subnet mask.

Return values

Return values	Description
SUCCESS	Route was successfully deleted from the table
DELETE_ROUTE_ERROR	Route was not deleted from the table

PPPGetPeerAssignedAddress

Gets the peer assigned IP Address of a PPP interface.

State

Callable from the `STARTED` state.

Format

```
int PPPGetPeerAssignedAddress (unsigned int commPort,
      unsigned long *ipAddress);
```

Arguments

Argument	Description
commPort	One of the following: ■ `PPP_COMPORT1` ■ `PPP_COMPORT2` This argument should match an interface already created by a `PPPCreateDevice` call.
ipAddress	Pointer to the variable storing the IP address.

Return values

Return values	Description
`SUCCESS`	IP address was retrieved successfully
`UNIT_OUT_OF_RANGE`	Port has not been created as a PPP port
`GET_ADDRESS_ERROR`	IP address was not retrieved

PPPModemClose

Hangs up the phone line, resets the modem, and closes the modem connection on the specified port.

State

Callable from the `STARTED` state and causes a transition to the `STOPPED` state.

Format

```
int PPPModemClose (unsigned int commPort);
```

Arguments

Argument	Description
commPort	One of the following: ■ `PPP_COMPORT1` ■ `PPP_COMPORT2` This argument should match an interface already created by a `PPPCreateDevice` call.

Return values

Return values	Description
`SUCCESS`	Phone was hung up, the modem was reset and the connection was closed
`UNIT_OUT_OF_RANGE`	Port has not been created as a PPP port
`MODEM_MISMATCH`	Modem is not connected to this serial port
`PORT_NOT_OPEN`	Port is not open
`CLOSE_PORT_ERROR`	Error closing port
`MODEM_NOT_ASSIGNED`	Modem's interface number has not been assigned Run `PPPModemInit` to assign interface number.

PPPModemInit

Opens the serial port to the specified baud rate, and initializes the modem.

State

Callable from the `STOPPED` state and causes a transition to the `STARTED` state.

Format

```
int PPPModemInit (unsigned int commPort, unsigned int mode,
        unsigned int baud, char *initString);
```

Arguments

Argument	Description
commPort	One of the following: ■ `PPP_COMPORT1` ■ `PPP_COMPORT2` This argument should match an interface already created by a `PPPCreateDevice` call.
mode	Determines who initiates the connection to the peer: ■ `PASSIVE_CONNECTION_MODE` — lets the peer start the connection ■ `ACTIVE_CONNECTION_MODE` — initiates a connection to the peer
baud	Baud rate for the serial port in the form `SIO_`*n*`_BAUD`, where *n* is one of the following: 75 7200 150 9600 300 14400 600 19200 1200 28800 2400 38400 4800 57600

Argument	Description
initString	Pointer to a buffer containing the modem initialization string. The string should be less than 64 bytes.

Return values

Return values	Description
SUCCESS	Success
UNIT_OUT_OF_RANGE	Port has not been created as a PPP port
OPEN_PORT_ERROR	Error opening serial port
SET_BAUD_ERROR	Error setting baud rate
INIT_MODEM_ERROR	Error initializing modem
INIT_STRING_TOO_LONG	Initialization string greater than 63 characters
SYSTEM_RESOURCE_ERROR	Could not start background thread for receive data
INVALID_CONNECTION_MODE	Error in specifying the connection mode
DIAL_ERROR	Failure to connect with active connection mode
LCP_SET_OPTIONS_ERROR	Error setting the default LCP protocol options

PPPSerialClose

Closes the directly connected serial port.

State

Callable from the STARTED state and causes a transition to the STOPPED state.

Format

```
int PPPSerialClose(unsigned int commPort);
```

Arguments

Argument	Description
commPort	One of the following: ■ PPP_COMPORT1 ■ PPP_COMPORT2 This argument should match an interface already created by a PPPCreateDevice call.

Return values

Return values	Description
SUCCESS	Success
UNIT_OUT_OF_RANGE	Port has not been created as a PPP port
PORT_NOT_OPEN	The port is not open
CLOSE_PORT_ERROR	Error closing port

PPPSerialInit

Opens and initializes the serial port to the specified baud rate.

State

Callable from the STOPPED state and causes a transition to the STARTED state.

Format

```
int PPPSerialInit (unsigned int commPort, unsigned int mode,
        unsigned int baud);
```

Arguments

<table>
<tr><th>Argument</th><th>Description</th></tr>
<tr><td>commPort</td><td>One of the following:
■ PPP_COMPORT1
■ PPP_COMPORT2
This argument should match an interface already created by a PPPCreateDevice call.</td></tr>
<tr><td>mode</td><td>Determines who initiates the connection to the peer:
■ PASSIVE_CONNECTION_MODE — lets the peer start the connection
■ ACTIVE_CONNECTION_MODE — initiates a connection to the peer</td></tr>
<tr><td>baud</td><td>Baud rate for the serial port in the form SIO_n_BAUD, where n is one of the following:
75   7200
150   9600
300   14400
600   19200
1200   28800
2400   38400
4800   57600</td></tr>
</table>

Return values

Return value	Description
SUCCESS	Success
UNIT_OUT_OF_RANGE	Port has not been created as a PPP port
OPEN_PORT_ERROR	Error opening serial port
SET_BAUD_ERROR	Error setting baud rate
SYSTEM_RESOURCE_ERROR	Could not start background thread for receive data
INVALID_CONNECTION_MODE	Error in specifying the connection mode
LCP_SET_OPIONS_ERROR	Error setting the default LCP protocol option

PPPSetAuth

Sets the authentication modes for the specified PPP interface using the CHAP or PAP protocols. Also sets the specified PPP interface's PAP username / password and CHAP ID / secret-key pairs.

Use `NAsetSysAccess` (in `sysAccess.h`) to add (or remove) peer PAP username / password pairs or CHAP ID / secret-key pairs to the internal table.

State

Callable from the `STOPPED` state.

Format

```
int PPPSetAuth(unsigned int commPort, char *papname,
        char *pappassword, char *chapname,
        char *chapsecret, int authentication);
```

Arguments

Arguments	Description
commPort	One of the following: ■ `PPP_COMPORT1` ■ `PPP_COMPORT2` This argument should match an interface already created by a `PPPCreateDevice` call
papname	PAP name of the specified PPP interface. After receiving a PAP authetication request, the PPP interface sends a PAP authentication reply where the name field in the reply is *papname*. This argument is ignored unless `ACCEPT_PAP` is specified for an *authentication* flag.

Arguments	Description
pappassword	PAP password of the specified PPP interface. After receiving a PAP authentication request, the PPP interface sends a PAP authentication reply where the password field is *pappassword*. This argument is ignored unless ACCEPT_PAP is specified for an *authentication* flag.
chapname	CHAP name of the specified PPP interface. After the PPP interface receives a CHAP authentication challenge, the PPP interface sends a CHAP response, where the name field of the response is the CHAP name provided by this argument. This argument is ignored unless ACCEPT_CHAP is specified for an *authentication* flag.
chapsecret	CHAP secret key of the specified PPP interface. After the PPP interface receives a CHAP authentication challenge, the PPP interface sends a CHAP response, where the encrypted string field of the CHAP response is generated using the CHAP secret key provided by this argument. Both PPP peers must share the same CHAP secret key. This argument is ignored unless ACCEPT_CHAP is specified for an *authentication* flag.
authentication	Authentication-protocol option initially requested during LCP options negotiation. The option bit flags are: ■ REQUIRE_CHAP ■ REQUIRE_PAP ■ ACCEPT_CHAP ■ ACCEPT_PAP See "*Authentication options*," below.

Authentication options

For the authentication type, the option flags can be bitwise ORed for more complex authentication operations, as follows:

Option flag	Description
REQUIRE_CHAP	Sets the specified PPP interface to require CHAP authentication from the peer.
REQUIRE_PAP	Sets the specified PPP interface to require PAP authentication from the peer.
REQUIRE_CHAP | REQUIRE_PAP	Sets the specified PPP interface to initially require CHAP authentication from the peer. If the peer declines CHAP and suggests PAP, the interface will then require PAP authentication.
ACCEPT_CHAP	Sets the specified PPP interface to accept CHAP authentication to the peer and refuse PAP.
ACCEPT_PAP	Sets the specified PPP interface to accept PAP authentication to the peer and refuse CHAP.
ACCEPT_PAP | ACCEPT_CHAP	Sets the specified PPP interface to accept CHAP or PAP authentication to the peer.

Return values

Return values	Description
SUCCESS	Success
UNIT_OUT_OF_RANGE	Port has not been created as a PPP port
INVALID_AUTHENTICATION	Unknown authentication type
NO_PASSWORD	No PAP password provided
NO_USERNAME	No PAP name provided
NO_HOSTNAME	No CHAP name provided
NO_SECRET	No CHAP secret key provided
ADD_USR_PWD_ERROR	The internal table is full or the username is null
LCP_SET_OPTIONS_ERROR	Error setting the default LCP protocol options
IPCP_SET_OPTIONS_ERROR	Error setting the default IPCP protocol options

PPPSetAuthentication

Sets the PPP interface to request or accept CHAP or PAP authentication. Sets the PAP username and password and / or CHAP ID and secret key.

Note: Deprecated function. This has been superseded by `PPPSetAuth`.

State

Callable from the `STOPPED` state.

Format

```
int PPPSetAuthentication(unsigned int commPort,
        char *papname, char *pappassword, char *chapname,
        char *chapsecret, int authentication);
```

Arguments

Arguments	Description
commPort	One of the following: ■ `PPP_COMPORT1` ■ `PPP_COMPORT2` This argument should match an interface already created by a `PPPCreateDevice` call.
papname	Username provided in response to a PAP authentication request. This name, along with the PAP password, is entered into the system database, which is used to authenticate a peer's reply to a PAP authentication request.
pappassword	Password provided in response to a PAP authentication request. This password, along with the *papname*, is entered into the system database, which is used to authenticate a peer's reply to a PAP authentication request.
chapname	CHAP name provided in response to a CHAP authentication request. This name, along with the *chapsecret*, is entered into the system database, which is used to authenticate a peer's reply to a CHAP challenge.

Arguments	Description
chapsecret	CHAP-encrypted value provided in response to a CHAP authentication request. This value, along with the *chapname*, is entered into the system database, which is used to authenticate a peer's reply to a CHAP challenge.
authentication	Authentication-protocol option initially requested during LCP options negotiation: ■ CHAP_AUTHENTICATION — request CHAP authentication ■ PAP_AUTHENTICATION — request PAP authentication ■ 0 — accept PAP or CHAP authentication

Return values

Return values	Description
SUCCESS	Success
UNIT_OUT_OF_RANGE	Port has not been created as a PPP port
INVALID_AUTHENTICATION	Unknown authentication type
NO_PASSWORD	No PAP password provided
NO_USERNAME	No PAP username provided
NO_HOSTNAME	No CHAP name provided
NO_SECRET	No CHAP secret key provided
ADD_USR_PWD_ERROR	The internal table is full or the username is null

PPPSetModemAutoAnswerRings

Sets the ring count before the modem answers an incoming call.

State

Callable from the `STOPPED` state.

Format

```
int PPPSetModemAutoAnswerRings (unsigned int phone_rings);
```

Description

Arguments

Arguments	Description
phone_rings	Number of rings before the modem answers the incoming call.

Return values

Return values	Description
`SUCCESS`	Success
`TOO_MANY_RINGS`	The phone rings more than 10 times

PPPSetModemDialString

Sets the modem dial string.

State

Callable from the `STOPPED` state.

Format

```
int PPPSetModemDialString (char *dial_string);
```

Arguments

Arguments	Description
dial_string	Pointer to the buffer storing the modem dial string.

Return values

Return values	Description
`SUCCESS`	Success
`DIAL_STRING_TOO_LONG`	The dial string is more than 31 characters

PPPSetPeerAddr

Specifies the IP address sent to the client through IPCP.

Use this function when you are using PPP as a server.

State

Callable from the `STOPPED` state.

Format

```
int PPPSetPeerAddr (unsigned int commPort,
        unsigned long ipAddress);
```

Arguments

Arguments	Description
commPort	One of the following: ■ `PPP_COMPORT1` ■ `PPP_COMPORT2` This argument should match an interface already created by a `PPPCreateDevice` call.
ipAddress	The IP address to send to the peer through IPCP.

Return values

Return values	Description
`SUCCESS`	Success
`UNIT_OUT_OF_RANGE`	Port has not been created as a PPP port

PPPSetVJ

Enables or disables Van Jacobson (VJ) compression or IP header compression.

By default, VJ compression is enabled.

State

Callable from the `STOPPED` state.

Format

```
int PPPSetVJ(unsigned int commPort, int require_VJ);
```

Arguments

Argument	Description
commPort	One of the following: ■ `PPP_COMPORT1` ■ `PPP_COMPORT2` This argument should match an interface already created by a `PPPCreateDevice` call.
require_VJ	Whether to use compression: ■ `DISABLE_VJ_COMPRESSION` ■ `ENABLE_VJ_COMPRESSION`

Return values

Return values	Description
`SUCCESS`	Compression was successfully enabled or disabled
`UNIT_OUT_OF_RANGE`	Port has not been created as a PPP port

Fast IP API

CHAPTER 9

Overview

Many applications need to respond quickly to requests sent by other devices on the network. Applications normally use the sockets API to receive request packets and send replies. The sockets API uses the following procedure:

1. When a request packet is received on the network, the Ethernet driver processes the receive packet and passes it to the IP stack.
2. The IP stack does more processing, and then passes the request to the application.
3. The application interprets the request and generates a response.
4. The application sends the packet to the IP stack, which processes it and sends it to the Ethernet driver to be transmitted.

The sockets API requires at least two thread switches and two buffer copies (of the request and reply packets). The entire process may take too long for some applications.

Why use fast IP?

Fast IP is an extension to the Ethernet driver and allows applications to bypass much of the processing described above if both the request and reply packets are UDP packets. Applications can use the fast IP API to have the Ethernet driver identify request packets and do one of the following:

- Send back a “canned” response.
- Call an application processing routine from the ISR to immediately process the packet and send back a reply.

All fast IP processing is performed within the Ethernet receive ISR. There are no thread switches, and buffer copies usually can be eliminated.

The exact processing time required by fast IP to receive a request and send back a reply varies from target to target, depending on the speed of the processor, the speed of the memory on the target, and the size of the request and reply packets.

Programming notes

Using fast IP has side effects. Because all fast IP processing is done within the Ethernet ISR, the ISR takes longer to complete — whether or not the received packet is a fast IP request packet. When fast IP request packets are received, the ISR must process them and send back a reply. This causes the ISR to run for a longer time. The result is that the amount of time that interrupts are disabled is lengthened when the Ethernet ISR is running, which may be unacceptable in some applications.

Applications that use fast IP must call either `fip_registerPortProcessingRtn` or `fip_registerPortAutoReply` to register IP port numbers with the ISR. Applications can associate a unique processing routine with each port, or have several ports share the same routine. Applications use the `fip_deregisterPort` function to discontinue processing.

Maximum number of fast IP ports

Fast IP maintains a table of the IP ports it services. The size of this table is determined by the `MAX_FIP_PORTS` constant defined in the `fast_ip.h` file.

This constant is set to 25.

Include file

Using the fast IP API requires the following header file:

```
fast_ip.h
```

Summary of fast IP functions

Function	Description
`fip_registerPortProcessingRtn`	Registers a fast IP port and an application-supplied processing routine that is called whenever the port received a UDP packet.
`fip_registerPortAutoReply`	Registers a fast IP port and a reply buffer for automatically replying to the sender of any packets received on that port.
`fip_deregisterPort`	Removes an IP port number and the associated processing routine or reply buffer from the table of fast IP ports maintained by the Ethernet driver.

Fast IP API functions

The following pages describe the fast IP API functions.

fip_registerPortProcessingRtn

Registers a fast IP port and an application-supplied processing routine. The Ethernet ISR will call the application's processing routine whenever it receives a UDP packet addressed to the indicated IP port.

You should open and bind a socket to the port number before calling this routine. This ensures that the port is not being used by another process.

Format

```
int fip_registerPortProcessingRtn (int fastPort, int flags,
        fip_handler *processingFn);
```

Arguments

Argument	Description
fastPort	IP port number to register with Ethernet ISR.
flags	Indicates how broadcast and multicast packets addressed to the fast IP port should be handled, and whether UDP checksums should be calculated for reply packets. The following bit definitions are used: ■ `FIP_ACCEPT_MULTICAST` — if multicast packets are to be passed up to the application's handler ■ `FIP_ACCEPT_BROADCAST` — if broadcast packets are to be passed up to the application's handler ■ `FIP_CALCULATE_UDP_CHECKSUM` — if UDP checksums are to be calculated for the *fastPort* For maximum performance, UDP checksums should not be calculated.
processingFn	Pointer to the application's handler for incoming packets.

Return values

Return value	Description
FIP_SUCCESS	Port number has been registered
FIP_PORT_ALREADY_REGISTERED	Port number was already registered
FIP_TABLE_FULL	Port table is full

Example

The following example shows how to use `fip_registerPortProcessingRtn` to install the `processPacket` function to be called whenever a packet received on IP port number 4000 is either a multicast packet or a unicast packet directed to this station. For optimum performance, UDP checksums are left disabled.

```
int flags = FIP_ACCEPT_MULTICAST;

if (fip_registerPortProcessingRtn (4000, FIP_ACCEPT_MULTICAST,
processPacket) != FIP_SUCCESS)
{
/* * * * code to handle error goes here * * */
}
```

processingFn

Passed as a parameter to `fip_registerPortProcessingRtn`. This function must be implemented in the application.

This handler is called directly by the Ethernet receive ISR. Therefore, it can use system services only available to ISRs, and can call functions only that also obey this restriction. If extensive processing of a packet is necessary, the function should schedule a thread to handle it.

Format

```
int processingFn (fip_requestStructure *request);
```

Arguments

Argument	Description
request	Pointer to the structure that contains information about the received packet.

fip_request *structure definition*

```
typedef struct
{
short unsigned requestPort;/* IP port number of request */
short unsigned replyPort;/* caller sets to dest port */
in_addr remoteAddress;   /* IP address of remote station*/
unsigned char *recvData;/* contents of data portion of
                           receive packet*/
int recvLength; /* length of data portion of receive packet*/
unsigned char *replyData;/* to be set to address of reply
                           packet */
int replyLength;         /* length of reply data*/
int replyMustBeCopied; /* set if reply data must be copied*/
} fip_requestStructure;
```

Fields in the fip_request *structure*

:

Field	Description
requestPort	Set to the destination IP port number of the receive packet.
replyPort	Before the Ethernet ISR calls the application's processing routine, the ISR loads this field with the source IP port number of the receive packet. If a reply is to be sent, then the application processing routine must load this field with the destination IP port number for the reply. In most cases, this will be the source IP port number of the request.
remoteAddress	IP address of the remote station that sent the packet.
recvData	Pointer to the data section of the incoming packet. This buffer belongs to the Ethernet ISR. The application must not write into this buffer after the processing routine returns, and the contents of this buffer may change immediately after the processing routine returns. If the application needs to save the contents of this buffer, it should set the `FIP_PASS_UP` bit in its return result so that the packet is passed onto the IP stack which will, in turn, pass the packet to the application through the standard sockets interface.
recvLength	Length of the data section of the incoming packet.
replyData	If a reply is to be sent, the application processing routine must load this field with the address of the data to be transmitted. The buffer must be addressable by DMA (must not be in cached memory). The buffer must be aligned on a 32-bit word boundary.
replyLength	Number of bytes in the buffer pointed to by *replyData*.

Field	Description
replyMustBeCopied	Set only if the contents of the reply buffer must be copied to an intermediate transmit buffer.If the buffer does not need to be copied, set this field to 0. The Ethernet driver uses DMA to transfer packets to the Ethernet transmitter. The DMA system runs in parallel with the CPU. Therefore, it is possible for packets to be transferred while another process is updating the packet buffer, even if interrupts are disabled. The DMA system transfers the packet 32 bits at a time, from the beginning of the buffer to the end. If the contents of the buffer are static, or if the contents are always valid even while being updated, set *replyMustBeCopied* to 0 for maximum performance. In this case, the application's reply buffer will be accessed with DMA directly to the Ethernet transmitter. If it is possible for the contents of the buffer to be invalidated while the packet is being accessed with DMA, *replyMustBeCopied* must be set.

Return values

The function should return an integer bit field. Setting the `FIP_SEND_REPLY` bit causes the reply buffer supplied by the application to be sent to the station that sent the request packet.

Setting the `FIP_PASS_UP` bit causes the receive packet to be passed to the IP stack. This can be useful if the application needs to have a thread process the packet. If the bit is not set, the packet will be discarded.

fip_registerPortAutoReply

Registers a fast IP port and a reply buffer for automatically sending the contents of that buffer to the sender of any packets received on the specified port.

You should open and bind a socket to the port number before calling this routine. This ensures that the port is not being used by another process.

Format

```
int fip_registerPortAutoReply (int fastPort, int flags,
      fip_replyType *reply);
```

Arguments

Argument	Description
fastPort	IP port number to register with Ethernet ISR.
flags	Indicates how broadcast and multicast packets that are addressed to the fast IP port should be handled, and whether UDP checksums should be calculated for reply packets. The following bit definitions are used: ■ FIP_ACCEPT_MULTICAST — if multicast packets are to be passed up to the application's handler ■ FIP_ACCEPT_BROADCAST — if broadcast packets are to be passed up to the application's handler ■ FIP_CALCULATE_UDP_CHECKSUM — if UDP checksums are to be calculated for the *fastPort* For maximum performance, UDP checksums should not be calculated.
reply	Pointer to information about the reply to be sent when packets are received on the selected IP port.

fip_ReplyType structure definition

```
typedef struct
{
   short unsigned replyPort;   /* port to send reply to */
   unsigned char *replyData;   /* to be set to address of
                                  reply packet */
   int replyLength;         /* length of reply data*/
   int replyMustBeCopied;   /* set if reply data must
                              be copied*/
} fip_replyType;
```

Fields in the fip_ReplyType *structure*

Field	Description
replyPort	Set to the IP port number to which the reply should be sent. Set to 0 (zero) to send the reply back to the port on the remote device that sent the request.
replyData	Pointer to the buffer with reply data. The buffer must be addressable by DMA (must not be in cached memory). The buffer must be aligned on a 32-bit word boundary.
replyLength	Number of bytes in the buffer pointed to by *replyData*.
replyMustBeCopied	This field should be set if the contents of the reply buffer must be copied to an intermediate transmit buffer. Set this field to 0 (zero) if the buffer does not need to be copied. The Ethernet driver uses DMA to transfer packets to the Ethernet transmitter. The DMA system runs in parallel with the CPU. Therefore, it is possible for packets to be transferred while another process is updating the packet buffer, even if interrupts are disabled. The DMA system transfers the packet 32 bits at a time, from the beginning of the buffer to the end.

Field	Description
	If the contents of the buffer are static, or if the contents are always valid even while being updated, set *replyMustBeCopied* to 0 for maximum performance. In this case, the application's reply buffer will be accessed with DMA directly to the Ethernet transmitter. If it is possible for the contents of the buffer to be invalidated while the packet is being accessed with DMA, *replyMustBeCopied* must be set.

Return values

Return value	Description
FIP_SUCCESS	Port number has been registered
FIP_PORT_ALREADY_REGISTERED	Port number was already registered
FIP_TABLE_FULL	Port table is full

fip_deregisterPort

Removes an IP port number and the associated processing routine or reply buffer from the table of fast IP ports maintained by the Ethernet driver.

Format

```
int fip_deregisterPort (int requestPort);
```

Arguments

Argument	Description
requestPort	IP port number to be deregistered.

Return values

Return value	Description
FIP_SUCCESS	Port number has been de-registered
FIP_PORT_NOT_REGISTERED	Port number was not registered

Example

This example shows how to use `fip_deregisterPort` to stop servicing packets on an IP port number. Once the function is called, fast IP will no longer intercept an IP packet with the indicated port number. In this example, you deregister port 6000.

```
fip_deregisterPort (6000);
```

Fast IP online examples

Two fast IP examples are provided in the BSP `apps` directory as follows:

- The first example demonstrates `fip_registerPortAutoReply` and `fip_registerPortProcessingRtn`.
- The second example contains test code to evaluate the non-fast IP UDP stack response time.

Each application has its own directory (`\src\examples\`) containing a `README` file which briefly explains the application.

DHCP API

CHAPTER 10

Overview

The Dynamic Host Configuration Protocol (DHCP) provides a way to dynamically allocate IP addresses to devices on the network.

To use the DHCP library, you must have a DHCP/BOOTP server on your network.

You can use DHCP to configure the following:

- IP address
- Default gateway
- Subnet mask
- Domain name server (DNS) list

How to enable DHCP

There are two ways to enable DHCP at startup:

- Use the application's `appconf.h` file
- Setting `APP_USE_NVRAM`

In either case, when DHCP is enabled during the BSP startup — specifically, the TCP / IP stack startup — the unit will generate datagrams conforming to RFC 1541. Datagrams are also generated (per the RFC) for IP address lease extension as needed.

To enable DHCP using the appconf.h *file*

1. Clear the `APP_USE_NVRAM` symbol (to 0).
2. Set the `APP_IP_USE_DHCP` symbol (to 1).

To enable DHCP using NVRAM

1. Set `APP_USE_NVRAM` to 1.
2. Use the `useDHCP` data member and enable / disable DHCP in the default parameters record (`defaultParams`) or from NVRAM (`nvParams`).

After obtaining an IP address, the DHCP client does the following:

- Configures the TCP / IP protocol stack with the correct IP address, subnet mask, and default gateway
- Starts a background task that renews the lease whenever it is about to expire
- Loads any DNS server acquired through DHCP into the list of known DNS servers
- Calls the `DhcpNowBound` public function

Once an IP address has been leased, the lease must be renewed periodically to prevent the DHCP server from assigning the IP address to another device on the network. If the lease ever expires, then the unit loses the right to use the IP address.

The DHCP client automatically tries to renew its IP address periodically to prevent the lease from expiring. If the lease cannot be renewed, the client calls the `DhcpLostLease`function. The application must then stop using that IP address. In practice, the only way to recover from this situation is to restart the unit and obtain a new IP address.

Summary of DHCP API functions

The DHCP client makes two calls into public functions (stub routines) that can be modified for customization. You can customize these stub routines to perform safety checks, re-register IP data, and so forth.

Function	Description
`DhcpNowBound`	Gets an IP address lease from a DHCP server.
`DhcpLostLease`	Shuts down critical services when there is a failure to extend an IP address lease from a DHCP server.

These stub routines are in the `dhcpstub.c` file.

DHCP API functions

The following pages describe the DHCP API functions.

DhcpNowBound

Called when the local DHCP client obtains an IP address lease from a DHCP server.

You can customize this stub routine to support specialized processing due to obtaining or changing IP parameters with DHCP. Call this routine after the DHCP ACK packet is received from the DHCP server.

Format

```
void DhcpNowBound (unsigned long ipAddress,
        unsigned long mask, unsigned long gateway);
```

Arguments

Arguments	Description
ipAddress	Unit’s IP address (in host byte order) obtained through DHCP.
mask	Unit’s subnet mask (in host byte order) obtained through DHCP.
gateway	IP address (in host byte order) of the unit’s default router obtained through DHCP.

Return values

none

DhcpLostLease

Called when the local DHCP client fails to extend an IP address lease from a DHCP server.

Failing to extend a lease is undesirable because the unit will not have an IP address and will be limited to UDP broadcasts only. Often, the failure to extend a lease also indicates some fatal network error, such as when the DHCP server is down.

You can use this routine to gracefully shutdown critical services that require constant network communication.

Format

```
void DhcpLostLease (void);
```

Arguments

none

Return values

none

Telnet API

CHAPTER 11

Overview

The Telnet protocol allows workstations to connect to a remote host and function as terminals hard-wired to the host.

The NET+OS Telnet server supports two ports, each with as many as 10 sessions running simultaneously. See the section on how Telnet sessions work, later in this chapter.

Include files

Using the the Telnet API requires the following header file:

```
tservapi.h
```

Also, you must link the following library in your build file:

```
na2.lib
```

Summary of Telnet API functions

Function	Description
TSInitServer	Initializes the Telnet server.
TSOpenPort	Configures the Telnet server to listen on a specific port.
Callback routines defined in the TSOpenPort structure	
connect_callback	Called when the Telnet server receives a connection request from a client.
login_callback	Called when the Telnet server accepts the client connection.
validate_callback	No longer used — the value in the TS_UCONFIG_TYPE structure will be ignored. Instead of using this routine, you should populate the system access database by calling NAsetSysAccess (in sysAccess.h).
disconnect_callback	Called when the Telnet server disconnects from the client. The user can activate the application process.
receive_callback	Called when the Telnet server receives incoming data.
getbuf_callback	Called when the Telnet server needs a buffer to store incoming data from the client.
TSClosePort	Closes the Telnet service at a specified port.
TSSetServerOption	Sets options for the Telnet server.
TSSetClientOption	Sets options for the Telnet client.
TSSendData	Sends data to a Telnet client.
TSSendString	Sends a text string to a Telnet client.
TSCloseSession	Closes a Telnet session.
TSGetSessionUsername	Returns a pointer to the username of the active session.

How Telnet sessions work

Multiple Telnet servers can be running simultaneously, polling for client connections for a specific port. Once a client connection is established, the server handles the negotiation of options.

After the negotiations are settled, the server executes the connect callback routine supplied by the developer. At this point, the user can implement code to initialize local data structures.

Next, the server requests a username and password. Once the client enters the login data, the username and password are verified by the server.

If the username and password are correct, the server executes the login callback function. At this point, the user can start the local processes.

When the connection is fully established, the server calls the receive callback function for any client data received.

The Telnet API provides functions to send data back to the client session.

After the client has sent all its data, the server executes the disconnect callback routine. At this point, the user can clean up local data structures and inform the local process that a close connection is pending.

Telnet and management variables

If an application needs to send the value of a management variable, then it must use one of the `manGet`*`xxx`* functions provided by the management API to read the data before sending it out.

For example, suppose the serial number for a device is stored as a character string in a management variable, and that the variable's ID is `serialNumber`. In such a case, the following code could be used to construct a string with the serial number and send it to the Telnet client:

```
writeWelcomeBanner (unsigned long sessionId)
{
   MAN_ERROR_TYPE ccode;
   int telnetError;
   char serialNumber[SERIAL_NUMBER_LENGTH+1];
   ccode = manGetArray ("serialNumber", serialNumber,
         sizeof(serialNumber), MAN_TIMEOUT_FOREVER, NULL);
      if (ccode != MAN_SUCCESS)
         {
               /* handle error */
         }
   TSSendString(sessionId, "Telnet server ");
   TSSendString(sessionId, serialNumber);
   TSSendString(sessionId, " welcomes you!\r\n\nWhat is your
pleasure? ");
}
```

Similarly, if the application receives data that needs to be stored in a management variable, it must use one of the `manSet`*xxx* functions provided by the management API to update the variable.

For example, suppose that the baud rate for a serial port is stored as a `WORD32` in the management variable called `baudRate`, and that the Telnet server prompts the user to enter a new baud rate value. In such a case, the following code could be used to update the management variable:

```
int applicationReceiveFunction(unsigned int sessionId,
         void *buf, int len)
   {
      static char buffer[MAX_BUFFER_LENGTH];
      int length;
               ....
code to build up buffer with complete line of data
               ....
   switch (serverState)
      {
         case RECEIVE_FIRST_NAME:
            updateFirstName(sessionId, buffer, length);
            promptForLastName(sessionId);
            serverState = RECEIVE_LAST_NAME;
            break;
                  ....
```

```
            other states
                  ....
      }
   }
void updateFirstName (unsigned long sessionId, char *buffer,
         int length)
   {
      MAN_ERROR_TYPE ccode;
         if (length > FIRST_NAME_LENGTH)
         {
            length = FIRST_NAME_LENGTH;
         }
         buffer[FIRST_NAME_LENGTH] = 0;
   ccode = manSetArray ("firstName", buffer,
      (FIRST_NAME_LENGTH + 1) * sizeof(char),
      MAN_TIMEOUT_FOREVER, NULL);
      if (ccode != MAN_SUCCESS)
         {
            /* handle error */
         }
   }
```

In this example, the `applicationReceiveFunction` is the callback function called by the Telnet server when it receives data from the client. It concatenates the data buffers until it has a complete line of data. Then it looks at an internal state variable to determine how to process the data. The `updateFirstName` function is called when the user has typed a first name. It stores it in the management variable.

Telnet API functions

The following pages describe the Telnet API functions.

TSInitServer

Initializes the Telnet server by specifying the maximum number of ports to be allocated. Multiple servers may be required when there is more than one port to be polled.

Format

```
int TSInitServer(TS_ICONFIG_TYPE *iconfig_ptr);
```

Arguments

Argument	Description
iconfig_ptr	Pointer to initialized configuration settings. See the structure definition below.

The `TS_ICONFIG_TYPE` structure is defined as follows:

```
typedef struct
{
    int max_entries;
}TS_ICONFIG_TYPE;
```

Return values

```
SUCCESS
TS_NO_MEMORY
TS_INVALID_PARAMETER
TS_SYSTEM_ERROR
TS_INVALID_STATE
```

TSOpenPort

Configures the Telnet server to listen on a specific port and other options.

Format

```
int TSOpenPort(unsigned long *port_id,
        TS_UCONFIG_TYPE *uconfig_ptr);
```

Arguments

Argument	Description
port_id	Number in the range — 0 to (*n–1)* where *n* is the number of entries defined in `TSInitServer`. This number is returned with a port number when called.
uconfig_ptr	Pointer to the user configuration options.
TS_UCONFIG_TYPE parameters (see structure definition below)	
telnet_port	Port number that the Telnet server is polling for incoming connections.
login_retries	Number of retries for the correct username and password match before the session is disconnected.
server_options	Options to negotiate for the Telnet server: ■ TS_ECHO ■ TS_TXBINARY ■ TS_NOGA
client_options	Options to negotiate for the Telnet client: ■ TS_ECHO ■ TS_TXBINARY ■ TS_NOGA
option_flag	Reserved.
connect_callback	Called when the Telnet server receives a connection request from a client.

Argument	Description
login_callback	Called when the Telnet server accepts the client connection.
disconnect_callback	Called when the Telnet server disconnects from the client.
receive_callback	Called when the Telnet server receives incoming data.
getbuf_callback	Called when the Telnet server needs a buffer to store incoming data from the client.

The TS_UCONFIG_TYPE structure is defined as follows.

```
typedef struct
{
   unsigned long telnet_port;
   int login_retries;
   unsigned long server_options;
   unsigned long client_options;
   unsigned long option_flag;
   int (*connect_callback) (unsigned int session_id);
   int (*login_callback)(unsigned int session_id, int);
   int (*validate_callback) (unsigned int session_id,
       char *username,char *password, int *uid);
   int (*disconnect_callback) (unsigned int session_id);
   int (*receive_callback) (unsigned int session_id,
        void *buf, int len, int status);
   int (*getbuf_callback) (unsigned int session_id,
        void **buf, int *size);
} TS_UCONFIG_TYPE;
```

Return values

SUCCESS
TS_INVALID_PARAMETER
TS_NO_MEMORY
TS_PORT_UNAVAILABLE
TS_INVALID_STATE

connect_callback

Called when the Telnet session receives a connection request from a client.

You can implement code to initialize local data structures.

Format

```
int (*connect_callback)(unsigned int session_id);
```

Arguments

Argument	Description
session_id	Unique identifier of the Telnet session.

Return values

Argument	Description
0	Success
-1	Notifies the server to close a session

login_callback

Called when the Telnet server accepts the client-requested connection. The user can activate the application process.

Format

```
int (*login_callback)(unsigned int session_id, int user_id);
```

Arguments

Argument	Description
session_id	Unique identifier of the Telnet session.
user_id	Identifier associated with the username when a session is validated during login.

Return values

Return values	Description
0	Success
1	Notifies the Telnet server to close a session

disconnect_callback

Called when the Telnet server disconnects the client connection.

Format

```
int (*disconnect_callback)(unsigned int session_id);
```

Arguments

Argument	Description
session_id	Unique identifier of the Telnet session.

Return values

Return values	Description
0	Success
1	Notifies the Telnet server to close a session

getbuf_callback

Called when the Telnet server needs a buffer to store incoming data from the client.

A default buffer allocation routine is provided if this function is not defined.

Format

```
int (*getbuf_callback)(unsigned int session_id, void **buf,
        int *buf_size);
```

Arguments

Argument	Description
session_id	Unique identifier of the Telnet session.
buf	Pointer to the buffer to store data.
buf_size	Size of the buffer. Default is 256 bytes.

Return values

Return values	Description
0	Success
1	Notifies the Telnet server to close a session

receive_callback

Called when the Telnet server receives incoming data from a client session.

Format

```
int (*receive_callback)(unsigned int session_id, void *buf,
        int len, int status);
```

Arguments

Argument	Description
session_id	Unique identifier of the Telnet session.
buf	Pointer to the buffer to store the data to be processed by the application.
len	Number of characters in the buffer.
status	Status of data input: ■ 1 — finished ■ 0 — unfinished

Return values

Return values	Description
0	Success
1	Notifies the Telnet server to close a session

TSClosePort

Closes the Telnet service for a specified port.

Format

```
int TSClosePort (unsigned long port_id);
```

Arguments

Argument	Description
port_id	Number in the range: 0-(*n*-1) where *n* is the number of entries defined in TSInitServer.

Usage notes

Before calling this function, make sure all active sessions are closed at that port. If there are active sessions, use TSCloseSession to close them. Otherwise, TSClosePort will fail (return TS_INVALID_STATE).

Also, you should call TSClosePort in different threads from the ones running on the specified port. This is to avoid deadlock. Therefore, do not call TSClosePort in the callback routines described above; instead, call it from the root routine when you want to close the port service.

Return values

SUCCESS
TS_INVALID_STATE
TS_INVALID_PARAMETER

TSSetServerOption

Sets options for the Telnet server.

Format

```
int TSSetServerOption (unsigned long session_id,
        unsigned long option, int mode);
```

Arguments

Argument	Description
session_id	Unique identifier of the Telnet session. Established during a call to `connect_callback`.
option	One of the following: ■ `TS_TXBINARY` — whether to transmit binary data ■ `TS_ECHO` — whether to echo characters ■ `TS_NOGA` — whether to suppress go-ahead characters ■ `TS_CAMOUFLAGE` — whether to echo an asterisk (*) for each character of a typed username and password
mode	0 = OFF 1 = ON

Return values

```
SUCCESS
TS_NO_SESSION
TS_SYSTEM_ERROR
TS_INVALID_PARAMETER
```

TSSetClientOption

Sets options for the Telnet client.

Format

```
int TSSetClientOption (unsigned long session_id,
        unsigned long option, int mode);
```

Arguments

Arguments	Description
session_id	Unique identifier of the Telnet session.
option	One of the following: ■ TS_TXBINARY — whether to transmit binary data ■ TS_ECHO — whether to echo characters ■ TS_NOGA — whether to suppress go-ahead characters
mode	0 = OFF 1 = ON

Return values

SUCCESS
TS_NO_SESSION
TS_SYSTEM_ERROR
TS_INVALID_PARAMETER

TSSendData

Sends data to a client through an established Telnet connection.

Format

```
int TSSendData (unsigned long session_id, char *buf,
        int len);
```

Arguments

Arguments	Description
session_id	Unique identifier of the Telnet session.
buf	Pointer to the buffer containing the data to be sent.
len	Size of the data in the buffer.

Return values

```
SUCCESS
TS_NO_SESSION
TS_SYSTEM_ERROR
TS_INVALID_STATE
TS_INVALID_PARAMETER
```

TSSendString

Sends a text string to a client through an established Telnet connection.

Format

```
int TSSendString (unsigned long session_id, char *buf);
```

Arguments

Arguments	Description
session_id	Unique identifier of the Telnet session.
buf	Pointer to the buffer containing the string to be sent. The buffer must be `NULL`-terminated.

Return values

SUCCESS
TS_NO_SESSION
TS_SYSTEM_ERROR
TS_INVALID_STATE
TS_INVALID_PARAMETER

TSCloseSession

Allows an external process to close a specific Telnet session.

Format

```
int TSCloseSession (unsigned long session_id);
```

Arguments

Argument	Description
session_id	Unique identifier of the Telnet session.

Return values

```
SUCCESS
TS_NO_SESSION
TS_SYSTEM_ERROR
```

TSGetSessionUsername

Returns a pointer to the username associated with the active session.

Format

```
int TSGetSessionUserName (unsigned long session_id);
```

Arguments

Argument	Description
session_id	Unique identifier of the Telnet session.

Return values

Argument	Description
NULL	Invalid or inactive session ID
string	Pointer to the username string

Sockets API

CHAPTER 12

Overview

The NET+OS sockets API is based on Berkeley Sockets, but is not fully compliant. The sockets API lets applications send and receive data over TCP/IP networks. Applications create communications endpoints (called *sockets*) with the `socket` function.

Endpoint address

In the TCP/IP protocol, the endpoint address consists of an IP address and a protocol port number. When a socket is created it has no endpoint address. A socket needs to have an endpoint address associated with it before the first data transfer.

An endpoint address can be associated with the socket in one of the following ways:

- Making a `bind` call — typically done for servers, which use a well-known protocol port number
- Making a `connect` or a first `sendto` call without a prior `bind` call

  In this case TCP/IP stack generates a port number and assigns it to a socket. This is typically done for clients that do not care what protocol port numbers they use.

One endpoint address is used for a socket lifetime.

Types of data transfer

The sockets API supports two types of data transfer — UDP and TCP.

UDP (SOCK_DGRAM)

UDP transmits individual packets between devices on the network. The UDP protocol does not guarantee that packets will arrive or that they will arrive in order. However, UDP is faster than TCP.

Applications can use the `sendto` and `recvfrom` functions to send and receive UDP packets. The `closesocket` function frees the UDP socket resource.

TCP (SOCK_STREAM)

TCP establishes connections between applications on the network. Once a connection between two applications is established, the applications can send any amount of data to each other. TCP protocol guarantees that the data will arrive or an error will be reported, and that the data will arrive in the order transmitted.

When establishing a TCP connection, one partner passively listens for a connection request, while the other partner actively attempts to connect. The `listen` and `accept` functions are used on the passive partner for listening and accepting a connection request. The `connect` function is used by the active partner to establish a connection. The `send` and `recv` functions are used by both partners to send and receive data. The `closesocket` function breaks a connection and frees the socket resource.

Fast sockets (fast UDP)

Also included is a nonstandard form of high-speed data transfer — fast sockets (fast UDP), based on the Berkeley standard.

The fast socket API transfers UDP datagrams as quickly as possible by using a sockets-like API that bypasses most of the TCP/IP stack. Data can be streamed out of the unit at potentially double the rate of the sockets API. The disadvantage of using this API is portability.

sockaddr_in structure

The `sockaddr_in` structure is used in several calls to specify or return a TCP/IP endpoint address. The structure is defined as follows:

```
   struct sockaddr_in {
   short sin_family;            /* must be AF_INET */
   unsigned short sin_port; /* 16-bit protocol port number */
   struct in_addr sin_addr;     /* 32-bit IP address */
   char sin_zero[8];            /* must be 0 */
};
```

Where:

- *sin_family* should be set to `AF_INET` (see `socket.h`)
- *sin_port* is the 16-bit service port number
- *sin_addr* is the 32-bit IP address
- *sin_zero[8]* is the 8 bytes of zeros, not currently used

All data fields are in network byte order.

Include files

The following header files are required:

- For the sockets API:

  `sockapi.h`

- For the fast sockets API:

  `fsock.h`

Summary of sockets API functions

Function	Description
accept	Accepts an incoming connection on a passive socket.
bind	Binds an endpoint address to a socket.
closesocket	Closes a socket.
connect	Initiates a connection.
getpeername	Gets the endpoint address of a connected peer.
getsockname	Gets the local endpoint address.
getsockopt	Gets socket options.
listen	Puts a socket in a passive listener mode.
recv	Receives data from a connected peer.
recvfrom	Receives data from a specified endpoint address.
select	Checks the status of multiple sockets.
send	Sends data to a connected peer.
sendto	Sends data to a specified endpoint address.
setsockopt	Sets sockets options.
shutdown	Terminates one or both directions of a full-duplex connection.
socket	Creates a socket.

Summary of fast socket API functions

Function	Description
fSockInit	Initializes and allocates a table of file descriptors.
fSocket	Creates a sock by calling the system function socket.
fBind	Binds an endpoint address to a socket.
fGetBuff	Returns a pointer to a buffer.
fSendTo	Sends a UDP packet.
fSocketClose	Closes a socket descriptor.

Sockets API functions

The following pages describe the sockets API functions.

Throughout the descriptions, socket descriptors are of type `int`, but can be substituted by the type `socket` (similar to WinSock) in all cases.

accept

Accepts an incoming connection on a passive socket. Servers use `accept` with connection-oriented or stream (TCP) sockets.

Format

```
int accept (int s, struct sockaddr_in *addr,int *addrlen);
```

Arguments

Argument	Description
s	Passive listener socket for accepting a connection request.
addr	Pointer to a `sockaddr_in` structure that `accept` fills in with the endpoint address of the connected socket.
addrlen	On input, pointer to the size of the *addr* buffer. On output, it is filled in with the size in bytes of the `sockaddr_in` structure.

Usage notes

Before `accept` is called, the socket must be set up as a passive listener to receive connection requests by issuing the `listen` system call.

The `accept` function then does the following:

- Extracts the first connection request on the queue of pending connections
- Creates a new socket with the same properties as the original socket
- Completes the connection with the remote peer socket
- Returns a descriptor for the new socket

The new returned socket descriptor is used to read and write data to and from the remote peer socket; it is not used to accept more connections. The original socket remains open for accepting further connections.

If no pending connections exist on the queue and the socket is blocking, `accept` blocks the caller until a connection is present. If the socket is set up as a non-blocking socket and no pending connections are present on the queue, `accept` returns an error and the socket error is marked `EWOULDBLOCK`.

Upon return, `accept` stores the endpoint address of the connected socket in the specified socket address structure.

Return values

Return value	Description
`ENOTSOCK`	First argument is not a valid socket
`EINVAL`	Invalid argument, or the socket is not a passive listener
`EFAULT`	Address length is less than `sizeof(struct` *`sockaddr_in`*)
`EWOULDBLOCK`	No incoming connections on a non-blocking socket

See also

`bind` `connect` `listen` `select` `socket`

bind

Assigns (binds) a 32-bit IP address and a 16-bit protocol port number to a socket.

Format

```
int bind (int s, struct sockaddr_in *addr,int *addrlen);
```

Arguments

Argument	Description
s	Socket to which the address is bound.
addr	Pointer to a `sockaddr_in` structure that stores the attributes to be bound to the socket.
addrlen	Size in bytes of the *addr* structure.

Usage notes

To simplify address binding, you can specify the `INADDR_ANY` symbolic constant as a wildcard Internet address, so you need not know the local Internet address. This also makes programs more portable. This symbolic constant is interpreted as any valid address. This allows the socket to receive data regardless of its node's Internet address.

For example, if a socket is bound to `<INADDR_ANY, 10>` and resides on a node attached to networks 90.0.0.2 and 100.0.0.3, the socket can receive data addressed to `<90.0.0.2, 10>` or `<100.0.0.3, 10>`.

Return values

Return value	Description
`ENOTSOCK`	First argument is not a valid socket
`EINVAL`	Invalid argument, or the socket is not a passive listener
`EFAULT`	Address length is less than `sizeof(struct` *sockaddr_in*)
`EAFNOSUPPORT`	Address family not supported
`EADDRINUSE`	Endpoint address already used by another socket

closesocket

Shuts down all the full-duplex connections on the specified socket and discards the socket descriptor.

Format

```
int closesocket (int s);
```

Arguments

Argument	Description
s	Socket to be closed.

Usage notes

If the SO_LINGER option is not set, any data queued at the socket is discarded.

If SO_LINGER is set, the system tries to transmit the data, queued at the socket, until the data is delivered with the linger interval timeout. If the socket is blocking, the system blocks the closesocket call while it tries to send out queued data. If the socket is non-blocking, the closesocket call returns immediately with the 0 return code, but the socket descriptor is actually discarded only after all queued data is sent with the linger interval timeout.

Using socketclose instead of closesocket is not recommended, but is now macro-defined to call closesocket.

Return values

Return value	Description
ENOTSOCK	First argument is not a valid socket
EINVAL	Invalid argument, or the socket is not a passive listener

See also

socket   accept   setsockopt

connect

Establishes an association between a local socket and a remote passive listener socket.

Format

```
int connect (int s, struct sockaddr_in *addr,int *addrlen);
```

Arguments

Argument	Description
s	Local socket.
addr	Pointer to a `sockaddr_in` structure that contains the address of the remote passive listener socket.
addrlen	Size in bytes of the *addr* structure.

Usage notes

Generally, a stream socket connects only once. A datagram socket can use `connect` multiple times to change its association.

If a stream socket is specified, `connect` initiates a three-way handshake (sync-sync/ack-ack) to the foreign passive listener socket. The caller is blocked until a connection is established, unless the socket is non-blocking, which then causes the function to return an error and the socket error is marked `EINPROGRESS`.

If a datagram socket is specified, `connect` associates the socket with the remote endpoint address supplied. This address is used by future `send` calls to determine the datagram's destination. This is the only address from which datagrams can be received by a `recv` call.

Return values

Return value	Description
`ENOTSOCK`	First argument is not a valid socket
`EINVAL`	Invalid argument, or the socket is not a passive listener
`EFAULT`	Address length is less than `sizeof(struct` *`sockaddr_in`*)
`EAFNOSUPPORT`	Address family not supported
`EALREADY`	Already connected, or initialted the connection
`ECONNREFUSED`	Specified endpoint address is not waiting for connections
`EINPROGRESS`	Initiating the connection on non-blocking socket

See also

`accept` `closesocket` `getsockname` `select` `socket`

getpeername

Gets the endpoint address of the connected peer (the remote socket at the other end of the connection).

Format

```
int getpeername (int s, struct sockaddr_in *addr,
        int *addrlen);
```

Arguments

Argument	Description
s	Original socket.
addr	Pointer to a `sockaddr_in` structure that `getpeername` fills in with the address of the connected peer socket.
addrlen	On input, must point to the size of the *addr* buffer. On output, it is filled in with the size in bytes of the `sockaddr_in` structure.

Return values

Return value	Description
`ENOTSOCK`	First argument is not a valid socket
`EINVAL`	Invalid argument, or the socket is not a passive listener

See also

`accept` `bind` `getsockname` `socket`

getsockname

Gets the local endpoint address of the specified socket.

Format

```
int getsockname (int s, struct sockaddr_in *addr,
       int *addrlen);
```

Arguments

Argument	Description
s	Local socket.
addr	Pointer to a `sockaddr_in` structure that `getsockname` fills in with the address of the local socket.
addrlen	On input, must point to the size of the *addr* buffer. On output, it is filled in with the size in bytes of the `sockaddr_in` structure.

Return values

Return value	Description
ENOTSOCK	First argument is not a valid socket
EINVAL	Invalid argument, or the socket is not a passive listener

See also

`bind` `getpeername` `socket`

getsockopt

Gets the status of options associated with the specified socket.

Only socket level options are supported. See the "*Socket options*" section at the end of this chapter.

Format

```
int getsockopt (int s, int level, int optname, char *optval,
      int *optlen);
```

Arguments

Argument	Description
s	Local socket.
level	Only `SOL_SOCKET` is supported (see `socket.h`).
optname	The option to be queried, and uses a symbolic constant. The symbolic constants supported are provided at the end of this chapter and in `socket.h`.
optval	Pointer to a buffer where `getsockopt` stores the value for the requested option. For most options, an `integer` is returned in the buffer pointed to by *optval*. For a Boolean option, a non-zero integer means the option is set, and a 0 means the option is off.
optlen	On output, it contains the actual size of the value returned in the *optval* buffer. On input, it contains the size of the buffer.

Return values

Return value	Description
ENOTSOCK	First argument is not a valid socket
EINVAL	Invalid argument, or the socket is not a passive listener
ENOPROTOOPT	Option not supported

listen

Puts a specified socket into the passive listener mode to receive connections.

Applies only to sockets of type SOCK_STREAM.

Connection requests are queued on the socket and accepted with accept. The maximum length of the queue of pending connections must be specified. If a connection request arrives while the queue is full, the requesting client gets an ECONNREFUSED error.

Format

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

Arguments

Argument	Description
s	Local socket.
backlog	Maximum length of the queue of pending connections (also called *packet queue depth*). Recommended range is 1–5.

Return values

Return value	Description
ENOTSOCK	First argument is not a valid socket
EINVAL	Invalid argument, or the socket is not a passive listener
EOPNOTSUPP	Socket does not support passive listener mode

See also

accept connect socket

recv

Transfers received data to the application.

Format

```
int recv(int s, char *buf, int len, int flags);
```

Arguments

Argument	Description
s	Socket from which data is received.
buf	Pointer to the user buffer where the data is stored.
len	Size of the buffer in bytes.
flags	Specifies usage options and is the result of an OR operation performed on one or more of the following symbolic constants (defined in `socket.h`). Can also be set to 0. ■ `MSG_OOB` specifies that you want `recv` to read any out-of-band data present on the socket, rather than the regular in-band data. Used with stream sockets only. ■ `MSG_PEEK` specifies that you want `recv` to peek at the data present on the socket; the data is returned, but not consumed, so that a subsequent receive operation sees the same data.

Usage notes

- For a connected datagram socket, `recv` behaves the same as `recvfrom`. The endpoint address, specified in the `connect` call, is used.
- For a stream socket, `recv` copies whatever data is available at the socket to the user buffer and returns. `recv` never copies more than `len` bytes of data to the user buffer, but it can copy less, if less than `len` bytes are available.
- If blocking mode is used, `recv` blocks the caller if no data is available at the socket. The caller is unblocked when data is received.

- If the socket has been marked non-blocking, `recv` returns immediately whether data is received. If no data is available on the socket, the function call fails and the socket error is marked `EWOULDBLOCK`.
- The `recv` system call returns the number of bytes received. This value should always be checked, because this is the only way to detect the actual number of data bytes stored in the user buffer.

Return values

Return value	Description
0	For a stream socket, receive was shut down by either end of the connection For a datagram sockets, a datagram without data was received
`ENOTSOCK`	First argument is not a valid socket
`EINVAL`	Invalid argument, or the socket is not a passive listener
`EWOULDBLOCK`	No data received on a non-blocking socket
`ENOTCONN`	Socket is not connected

See also

`connect` `recvfrom` `socket`

recvfrom

Transfers received data to the application.

This function is almost identical to `recv`. The difference is `recvfrom` can also return the endpoint address of the sender in the specified parameter.

Format

```
int recvfrom(int s, char *buf, int len, int flags,
       struct sockaddr_in from, int fromlen);
```

Arguments

Argument	Description
s	Socket from which data is received.
buf	Pointer to the user buffer where data is stored.
len	Size of the buffer in bytes.
flags	Specifies usage options and is the result of an OR operation performed on one or more of the following symbolic constants (defined in `socket.h`). Can also be set to 0. `MSG_PEEK` specifies that you want `recvfrom` to peek at the data present on the socket; the data is returned, but not consumed, so that a subsequent receive operation sees the same data.
from	If this is not a `NULL` pointer, `recvfrom` fills in the `sockaddr_in` structure it points to with the address of the received data's sender.
fromlen	On input, pointer to the size of the *from* buffer. On output, it is filled in with the size of the *from structure*.

Return values

Return value	Description
`ENOTSOCK`	First argument is not a valid socket
`EINVAL`	Invalid argument, or the socket is not a passive listener
`EWOULDBLOCK`	No data received on a non-blocking socket

select

Used to multiplex I/O requests among multiple sockets.

Format

```
int select (int width, fd_set *readset, fd_set *writeset,
       fd_set *exceptset,struct timeval *timeout);
```

Arguments

Argument	Description
width	The largest socket descriptor plus 1.
readset	Pointer to a set of sockets from which to read.
writeset	Pointer to a set of sockets to which to write.
exceptset	Pointer to a set of sockets that may have an exceptional condition pending.

timeout	Pointer to the timeout interval. If the total timeout is less than 10 msec, `select` returns immediately.
	If the *timeout* is a `NULL` pointer, the maximum unsigned long integer is used.
	The `timeval` structure is defined in `ftype.h` as follows:
	`struct timeval`
	`{`
	`  long tv_sec;   /* number of seconds */`
	`  long tv_usec;} /* number of microseconds`
	`                  (not currently used)*/`
	`}`

Usage notes

Three sets of socket descriptors can be specified:

- A set from which to read
- A set to which to write
- A set that may have pending exceptional conditions

If `select` returns a positive number, the three sets indicate which socket descriptors can be read, which can be written to, or which have exceptional conditions pending. A timeout value can be specified.

Selecting for reading a socket descriptor upon which a `listen` call has been performed indicates that a subsequent `accept` call will not block.

The following standard entry points are provided for manipulating such descriptor sets:

- `FD_ZERO`
- `FD_SET`
- `FD_CLR`
- `FD_ISSET`

These entry points behave according to the standard Berkeley semantics.

The status of a socket descriptor in a set can be tested with the `FD_ISSET(`*s*`, &set)` macro, which returns a non-zero value if *s* is a member of the set, or 0 if it is not set.

In addition, the `FD_SET` and `FD_CLR` macros are provided for adding and removing socket descriptors to and from a set. The `FD_ZERO` macro is provided to clear the set and should be used before the set is used. These macros are defined in `select.h`.

The `FD_SETSIZE` constant (also in `select.h`) identifies the maximum value of the *width* argument.

Example

The following example shows how to use `select` to determine if two sockets have available data:

```
fd_set read_set;

struct timeval wait;

for (;;)
{
   wait.tv_sec = 1;      /* wait for 1 second */
   wait.tv_usec = 0;

   FD_ZERO (&read_set);
   FD_SET (s1, &read_set);
   FD_SET (s2, &read_set);

   nb = select (FD_SETSIZE, &read_set, (fd_set *) 0,
         (fd_set *) 0, &wait);
   if (nb <= 0)
   {
      /* error occurred or timed out */
   }

   if (FD_ISSET(s1, &read_set))
   {
      /* socket 1 has data available */
   }

   if (FD_ISSET(s2, &read_set))
   {
      /* socket 2 has data available */
   }
}
```

Note: If two tasks attempt to use `select` on the same socket for the same conditions, an error occurs.

Return values

Return value	Description
ENOMEM	Cannot allocate memory for internal tables
EINVAL	Invalid argument, or the socket is not a passive listener

See also

accept connect listen recv send

send

Sends data to a remote socket.

If no buffer space is available at the remote socket to hold the data to be transmitted (for example, if the TCP window is less than buffer size), `send` blocks the calling task unless the socket has been marked non-blocking. If the socket is set up as a non-blocking socket and no buffer space is available, the function returns an error and the socket error is marked `EWOULDBLOCK`.

Note: A network application can use `select` to determine when the socket will again be writable.

Format

```
int send(int s, char *buf, int len, int flags);
```

Arguments

Argument	Description
s	Local socket, which must be in a connected state. If *s* is a stream socket, the data is sent to the foreign socket that is connected to *s*. If *s* is a datagram socket, the data is sent to the socket that has been associated with *s* through a previous `connect` system call.
buf	Pointer to a buffer containing the data to send. If *s* is a datagram socket, the data that *buf* points to is a datagram.
len	Number of bytes in *buf*.
flags	Specifies usage options and is the result of an OR operation performed on one or more of the symbolic constants defined in `socket.h`. It can also be set to 0. Specify `MSG_OOB` to send out-of-band data, rather than the regular in-band data. This is only for stream sockets. Specify `MSG_DONTROUTE` to send data without routing tables.

Return values

Return value	Description
ENOTSOCK	First argument is not a valid socket
EINVAL	Invalid argument, or the socket is not a passive listener
ENOTCONN	Socket not connected
ECONNABORTED	Connection aborted by local socket
ESHUTDOWN	Socket output shut down

See also

`sendto` `socket`

sendto

Sends a datagram to a remote UDP socket.

Although it is possible to use this function with stream data, it is intended to be used only with datagram sockets.

Format

```
int sendto (int s, char *buf, int len, int flags,
       struct sockaddr_in *to, int tolen);
```

Arguments

Argument	Description
s	Local socket.
buf	Pointer to a buffer that contains the data to send.
len	The number of bytes in *buf*.
flags	Specifies usage options and is the result of an OR operation performed on one or more of the symbolic constants defined in socket.h. It can also be set to 0. Specify MSG_DONTROUTE to send data without routing tables.
to	The destination socket address — and a pointer to the sockaddr_in structure.
tolen	The size in bytes of the *to* argument.

Return values

Return value	Description
ENOTSOCK	First argument is not a valid socket
EINVAL	Invalid argument, or the socket is not a passive listener
EFAULT	Wrong endpoint address length
EAFNOSUPPORT	Address family not supported
EMSGSIZE	Datagram is larger than the socket send buffer

setsockopt

Sets options associated with the specified socket.

Only socket level options are supported. See the "*Socket options*" section at the end of this chapter.

Format

```
int setsockopt (int s, int level, int optname, char *optval,
        int optlen);
```

Arguments

Argument	Description
s	Socket whose options are to be set.
level	Only SOL_SOCKET is supported (see socket.h).
optname	Socket option to be set according to a symbolic constant defined in socket.h. The symbolic constants supported are described at the end of this chapter.
optval	Pointer to a buffer where the option value is specified. Most options are 32-bit values. A non-zero value for a Boolean option means the option should be set; a 0 means the option should be turned off.
optlen	Size of the value pointed to by *optval*.

Return values

Return value	Description
ENOTSOCK	First argument is not a valid socket
EINVAL	Invalid argument, or the socket is not a passive listener
ENOPROTOOPT	Option not supported

See also

accept   bind   closesocket   connect   listen   getsockopt

shutdown

Terminates all or part of a full-duplex connection on a specified socket. The socket can be shut down for sending, receiving, or both.

Format

```
int shutdown (int s,int how);
```

Arguments

Argument	Description
s	Socket to be shut down.
how	Specifies the shutdown mechanism: ■ 0 — No further receives are allowed on the socket ■ 1 — No further sends are allowed on the socket ■ 2 — No further sends or receives are allowed on the socket

Return values

Return value	Description
ENOTSOCK	First argument is not a valid socket
EINVAL	Invalid argument, or the socket is not a passive listener

socket

Creates a new socket and returns its socket descriptor.

Format

```
int socket (int domain, int type, int protocol);
```

Arguments

Argument	Description
domain	Socket domain and must be set to *AF_INET*.
type	Type of socket (as defined in `socket.h`): ■ `SOCK_STREAM` — defines a stream socket, which uses TCP to provide a reliable connection-based communication service ■ `SOCK_DGRAM` — defines a datagram socket, which uses UDP to provide a datagram service
protocol	Must be 0.

Return values

Return value	Description
ENOTSOCK	First argument is not a valid socket
EINVAL	Invalid argument, or the socket is not a passive listener
EAFNOSUPPORT	Address family not supported
EPROTONOSUPPORT	Protocol not supported
ESOCKNOSUPPORT	Socket type not supported

Fast socket (fast UDP) API functions

The following pages contain the fast socket (fast UDP) API functions.

fSockInit

Initializes and allocates a table of file descriptors.

Format

```
void fSockInit (int num_fd);
```

Arguments

Argument	Description
num_fd	Maximum number of file descriptors in the table.

Return values

Return value	Description
0	Success
-1	Unable to allocate space

fSocket

Creates a fast UDP socket for sending high-speed datagrams using `fGetBuff` and `fSendTo`.

Applications are limited to no more than 3 fast UDP sockets.

Format

```
SOCKET fSocket (int domain, int type, int protocol);
```

Arguments

Argument	Description
domain	Socket domain. Must be set to *AF_INET*.
type	Type of socket (defined in `socket.h`). Must be set to `SOCK_DGRAM` — datagram socket, which uses UDP to provide a datagram service
protocol	Must be 0.

Return values

Return value	Description
integer	Socket descriptor
-1	Error occurred

See also

`fGetBuff` `fSendTo`

fBind

Binds the fast UDP socket to a service port.

Format

```
long fBind (int s, struct sockaddr_in *addr,
        int addrlen);
```

Arguments

Argument	Description
s	Socket to which the address is bound.
addr	Pointer to a `sockaddr_in` structure.
addrlen	Size of a `sockaddr_in` structure.

Return values

Return value	Description
0	Success
-1	Failure

See also

`fSocket`

fGetBuff

Returns a pointer to fast UDP buffer used in an `fSendTo` call.

This buffer becomes the data payload encapsulated by a UDP header in an IP packet. The buffer can be used only with the fast UDP socket *(fd)*.

Format

```
char *fGetBuff (SOCKET fd, int szData);
```

Arguments

Argument	Description
fd	Socket with which to associate the buffer.
szData	Size of the buffer to allocate. Maximum 1470 bytes.

Return values

Return value	Description
integer	Address of the buffer
NULL	Error occurred — buffer exceeded the maximum Ethernet packet data size

See also

`fSocket` `fSendTo`

fSendTo

Sends a fast UDP packet, defined by the data in a buffer, to a specified address.

Format

```
long fSendTo (int fd, char *buf, int len, int flags
        struct sockaddr_in *sa,int tolen);
```

Arguments

Argument	Description
fd	Local socket.
buf	Pointer to a fast UDP buffer that contains the data to send.
len	Number of bytes in *buf*.
flags	If set to 1, turns off error checking in the routine.
sa	Pointer to the `sockaddr_in` structure.
tolen	Size of the *sa* data in bytes.

Return values

Return value	Description
integer	Number of bytes sent
-1	Error occurred

See also

`fSocket` `fGetBuff` `fSocketClose`

fSocketClose

Closes a socket and then frees any buffers still associated with the socket.

Format

```
void fSocketClose (int s);
```

Arguments

Argument	Description
s	Socket to be closed.

Return values

none

Socket options

The getsockopt and setsockopt calls use the following socket options.

Option	Description	get/set	Default
SO_ERROR	Returns the pending error and clears the error status.	get	n/a
SO_KEEPALIVE	Keeps the connection alive by periodically transmitting a packet over socket *s*.	get/set	off

SO_LINGER	Controls the action taken when unsent messages are queued on a socket and a socketclose is executed.
	If the socket is a stream socket and SO_LINGER is set (l_onoff set to 1), the calling task blocks until it can transmit the data or until a timeout period expires.
	If SO_LINGER is disabled (l_onoff set to 0), the socket is deleted immediately. SO_LINGER uses the linger structure, which is defined as follows:
	`struct linger`
	`{`
	`  int l_onoff;   /* on/off option */`
	`  int l_linger;  /* linger time in secs`
	`*/`
	`}`
		get/set	off

SO_MAXMSG	Returns the value of the TCP maximum segment size.	get	1458
SO_MYADDR	Returns the local IP address of the socket.	get	n/a
SO_NONBLOCK	Indicates/controls whether a socket is non-blocking. The **optval* parameter must point to an int. This int will be set to 0 (zero) if the socket has been set for blocking mode, or 1 if it is set for non-blocking mode.	get/set	Blocking I/O
SO_OOBINLINE	Requests that out-of-band data go into the normal data input queue as received; it then is accessible with recv calls without the MSG_OOB flag.	set	off
SO_RCVBUF	Adjusts the buffer size allocated for a socket input buffer. This size is related to the TCP window size.	get/set	

Option	Description	get/set	Default
SO_REUSEADDR	Indicates that local addresses can be reused in a bind call.	get/set	off
SO_RXDATA	Indicates the number of bytes of data in the socket receive buffer.	get	n/a
SO_SNDBUF	Adjusts the normal buffer size allocated for a socket output buffer.	get/set	
SO_TYPE	Returns the type of socket.	get	n/a
SO_BIO	Sets the socket to use blocking I/O.	set	n/a
SO_NBIO	Sets the socket to use non-blocking I/O.	set	n/a
SO_BROADCAST	Sets the socket send to send broadcast packet.	get/set	off

Level IPPROTO_IP socket options

Option	Description	get/set	Default
IP_ADD_MEMBERSHIP	Join a multicast group. This option uses the ip_mreq structure (see definition below). If the interface address is not specified, the default interface will be assumed.	get/set	n/a
IP_DROP_MEMBERSHIP	Leave a multicast group. This option uses the same ip_mreq structure defined in IP_ADD_MEMBERSHIP. If the interface address is not specified, the first interface that supports multicast will be assumed. Usually, this is the default interface.	set	n/a
IP_MULTICAST_LOOP	Loopback of outgoing multicast. Datagram to the upper protocol layer.	get/set	on
IP_MULTICAST_TTL	Datagram’s time-to-live field.	get/set	1

ip_mreq structure definition

The ip_mreq structure is defined as follows:

```
struct ip_mreq
{
   ip_addr imr_multiaddr;/* IP multicast address of group */
   ip_addr imr_interface;/* local IP address of interface */
};
```

Socket error codes

Code	Mnemonic	Description
11	EWOULDBLOCK EAGAIN	Operation would block on a blocking socket
12	ENOMEM	Cannot allocate memory
14	EFAULT	Invalid argument
22	EINVAL	Invalid argument, or operation or out-of-band data
32	EPIPE	The connection is broken
36	EINPROGRESS	Operation now in progress
37	EALREADY	Operation already in progress
39	EDESTADDRREQ	Destination address required
40	EMSGSIZE	Message too long
42	ENOPROTOOPT	Option not supported
43	EPROTONOSUPPORT	Protocol not supported
44	ESOCKTNOSUPPORT	Socket type not supported
45	EOPNOTSUPP	Operation not supported
47	EAFNOSUPPORT	Address family not supported
48	EADDRINUSE	Address is already in use
49	EADDRNOTAVAIL	Cannot assign requested address

Code	Mnemonic	Description
50	ENETDOWN	Network is down
51	ENETUNREACH	Invalid network location
53	ECONNABORTED	Software caused connection abort
54	ECONNRESET	Connection reset by peer
55	ENOBUFS	No buffer space available
56	EISCONN	Socket already connected
57	ENOTCONN	Socket not connected
58	ESHUTDOWN	Cannot perform an operation after socket shutdown
59	ETOOMANYREFS	Exceeds the maximum number of multicast memberships (20)
60	ETIMEDOUT	Operation timed out
61	ECONNREFUSED	Connection refused

HTTP Server API

CHAPTER 13

Overview

The HTTP server API lets you configure Web page access and page properties and define addresses and the page content.

Using the HTTP server API, you can:

- Send back data, such as an HTML page, to an individual application
- Provide an application with easy access to end-user supplied data sent back from Web browsers

For example, if a form is completed on the Web, and a user clicks a Submit button, the data can be sent back to an application.

Note: The HTTP server interface is not recommended or supported with large or multiple Web pages. Instead, use the Advanced Web Server interface (see chapter 2).

Include file

Using the HTTP server API requires the following header file:

`hservapi.h`

Also, you must add the `httpd.a` library to a project that uses the HTTP server.

Summary of HTTP server API functions

Functon	Description
`HSCustomPage`	Lets you customize your response page.
`HSGetValue`	Used by applications that process HTML form data.
`HSPageAccess†`	Controls Web page access via usernames and passwords.
`HSPageAuthenticate`	Controls Web page access.
`HSProperties`	Modifies task properties of the HTTP server.
`HSRegisterSearchFunction`	Supplies the HTTP serverwith the routine that matches browser requests with the content to be supplied by an application.
`HSRemoteAddress`	Retrieves the IP address of the requesting browser.
`HSSend`	Returns HTML page content to a requesting browser.
`HSSendBinary`	Enables the return of binary data, such as images and Java applets.
`HSSetRealm†`	Sets up the security realm’s name, username, and password.
`HSStartServer`	Enables an application to start the HTTP server.

Functon	Description
MIME content functions	
HSTypeHtml HSTypeGif HSTypeApplet HSTypeText HSTypeJpeg HSTypePict HSTypeTiff HSTypePng HSTypeXbm HSTypeWav HSTypeAu HSTypeStaticApplet HSTypeStaticGif HSTypeStaticHtml HSTypeStaticJpeg HSTypeStaticPict HSTypeStaticPng HSTypeStaticText HSTypeStaticTiff HSTypeStaticAu HSTypeStaticXbm	These functions set the MIME type for the component being returned by the HTTP server.

† = Deprecated function

Stub routines

The following stub functions are defined in the `url.c` file generated by the HTML-to-C Converter:

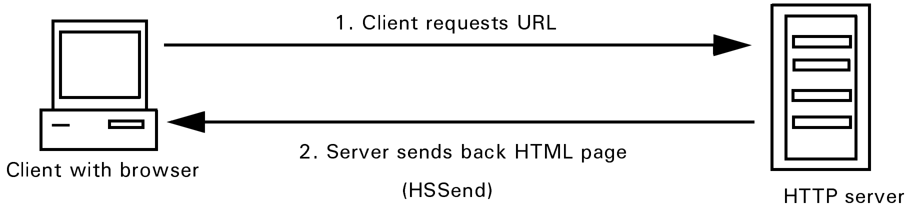
- `HSInitSecurityTable` — Sets up security information
- `SearchURL` — Application URL search routine

Understanding HTTP and HTML

The Internet connects two types of computers — servers which provide content, and clients which retrieve and display documents. These documents are displayed on the client side through the use of HTML.

To access and display the HTML documents, a user runs a browser on the client computer. These browser clients speak to special Web servers over the Internet to access and retrieve electronic documents.

A browser begins loading an HTML document either from local storage or from a network, such as the Internet. When using the Internet, the client browser first consults the domain name server to translate the document server's name, such as *www.netsilicon.com*, into an IP address before sending a request to that server over the Internet. This request and the server's reply is formatted according to the HTTP standard.



Forms

A form is a method for sending data to an HTTP server for processing.

You can place a form anywhere inside the body of an HTML document using the *<form>* and *</form>* HTML tags. You can include regular text to label user-input fields and to provide directions, for example.

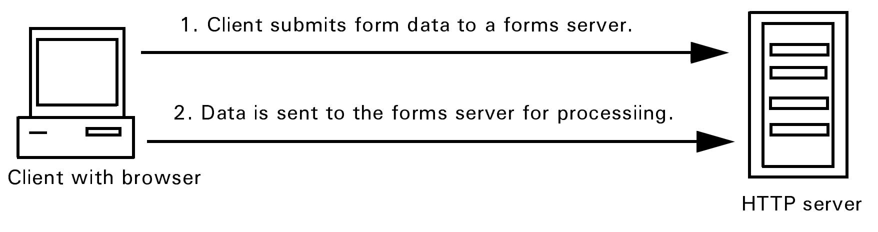
Browsers use the special form elements as if they were small images embedded in text. All form elements within the *<form>* and *</form>* tags comprise a single form and a browser sends all the values of these elements — blank, default, or user-modified — when a user submits a form to the server.

You normally define the form attributes that provide the name of the form's processing server and the method by which the parameters are to be sent to the server. You can also define an attribute that lets you change how the parameters get encoded for secure transmission over the network.

The POST method is the way the browser sends the form's data to the server for processing. With the POST method, a browser sends the data in two steps:

1. The browser first contacts the form-processing server.
2. Once contact is made, the browser sends the data to the server in a separate transmission. (See the illustration below.)

On the server side, the POST-style applications read the parameters from a standard location once execution begins. Once the parameters are read, the parameters must be decoded before the application can use the form values. A particular server will define exactly how the POST-style applications can expect to receive the parameters.



To process the forms data on a server, all server-side applications pass their results back to the server and on to the user by writing that result to the application's standard output as a MIME-encoded file.

Sample applications

Use the HTTP server API to do tasks such as:

- Implementing configuration menus as HTML pages

  For example, you can create pages to allow users to configure network devices from a Web browser.

- Creating general-purpose user interfaces to network devices, which can be implemented through the HTTP server

  For example, a device that controls factory automation equipment could be controlled through an HTML page accessible from a Web browser.

- Implementing debugging screens, error logs, and so on, as HTML pages

  You can hide these screens from the end user by giving the screens an undocumented pathname off the device's base URL.

HTML data types: functional description

This section gives a functional description that makes the server easier to use and to integrate with your own application-specific code. These functions can be used in any combination as part of your application.

Static content

One function an HTTP server must provide is the ability to return *static components*. A static component is an HTML page, image, or Java applet where the content does not change. A Web browser issues an HTTP GET request specifying a URL, and the HTTP server responds by returning the requested page, image, or applet.

Static components are added to the HTTP server by running the HTML-to-C Converter on HTML pages, images, or applets. The *NET+OS User's Guide* gives a detailed description of the HTML-to-C Converter.

Integrating the Web components into the HTTP server includes:

- Updating the NET+Works application makefile or build file to include the new C source code files
- Rebuilding the software

Dynamic content

Another function is to provide *dynamic content* that changes over time via an HTML page. For example, an application developer may want to return information that changes due to reconfiguration. The HTML pages in this case usually consist of some static information (defined above), combined with changing dynamic information. The HTTP server API allows for this dynamic content by using reserved keywords embedded within the HTML page.

The HTML-to-C Converter scans each HTML page for keywords with the prefix `_NZZA_`. Each keyword converts to a routine called from the routine responsible for returning the specified HTML page. Routines created by the HTML-to-C Converter are empty. An application developer must add code to perform any specific processing required and then return specified HTML page information based upon that processing. An example is given starting on the next page.

HTML processing code example

The following is an HTML-to-C code conversion example in which a device is assigned an IP address and the default gateway is defined:

HTML page: config.htm

```
<HTML><BODY>
<H1>Configuration Page</H1>
<B>IP Address </B>_NZZA_ip_address<BR>
<B>Default Gateway </B>_NZZA_def_gateway<BR>
</BODY></HTML>
```

HTML-to-C Converter partial output

```
void Send_function_0(unsigned long handle)
{
 HSSend (handle, “<HTML><BODY>”);
 HSSend (handle, “<H1>Configuration Page</H1>”);
 HSSend (handle, “<B>IP Address </B>”);
 na_ip_address (handle);
 HSSend (handle, “<BR>”);
 HSSend (handle, “<B>Default Gateway </B>”);
 na_def_gateway (handle);
 HSSend (handle, “<BR>”);
 HSSend (handle, “</BODY></HTML>”);
}
void na_ip_address (unsigned long handle)
{
}
void na_def_gateway (unsigned long handle)
{
}
```

In the example, an application developer must fill in the contents of `na_ip_address` and `na_def_gateway` to assign the IP address and default gateway address.

For example, the developer might complete the `na_ip_address` function as follows:

```
void na_ip_address(unsigned long handle)
{
unsigned long ip_address;
char display_string[16];
/*    call a function that returns IP address /*
/*    as an unsigned long */
ip_address = getMyIPAddress();
/*    convert unsigned long IP to dot-notation /*
/*    printable string /*
```

```
sprintf(display_string,     */
"%d.%d.%d.%d",
(ip_address >> 24) & 0xFF,
(ip_address >> 16) & 0xFF,
(ip_address >> 8) & 0xFF,
ip_address & 0xFF);
HSSend(handle, display_string);/* and serve it up */
}
```

Forms processing

The HTML-to-C Converter scans each HTML page for the `<FORM ACTION=”form_page” METHOD=”POST”>` elements.

The `<FORM>` tag invokes server side processing of data from the Web browser for a Web user. Each form construct is converted into a C routine (via the HTML-to-C Converter) that is called when form data is posted to the HTTP server. The application developer fills in this routine with code that processes the returned data.

The HTTP server API provides routines that allow accessing returned data. The `form_page` keyword is converted by the HTML-to-C Converter into a routine named `Post_form_page`, which is called when data is posted to the URL `form_page`.

The following is a code example of forms processing:

HTML page: forms.htm

```
<html><body>
<form action="my_post" method="POST">
<p><input type=submit name="Submit" value="Submit">
<input type=reset name="Reset" value="Reset">
<select name="Pull_down" size=1>
<option>Option1</option>
<option>Option2</option>
<option>Option3</option>
<option>Option4</option>
```

```
</select></p>
<p>Enter some text here : <input type=text size=20 maxlength=256
name="text_box"></p>
<p>Some more text here : <input type=text size=20 maxlength=256
name="more_text"></p>
</form>
</body></html>
```

Generated C code

Use the HTML-to-C Converter to generate the C code from the NETARM.EXE program.

The following are the results:

```
void Send_function_0(unsigned long handle)
{
 HSSend (handle, "<!DOCTYPE HTML PUBLIC \"-//IETF//DTD HTML/
EN\">");
 HSSend (handle, "<html>");
 HSSend (handle, "<head>");
 HSSend (handle, "<title>Untitled Normal Page</title>");
 HSSend (handle, "<meta name=\"GENERATOR\"
content="Microsoft FrontPage 1.1\">");
 HSSend (handle, "</head>");
 HSSend (handle, "</body>");
 HSSend (handle, "<form action=\"my_post\" method=\"POST\">");
 HSSend (handle, "<p><input type=submit name=\"Submit\"
value=\"Submit\">");
 HSSend (handle, "<input type=reset name=\"Reset\"
value=\"Reset\">");
 HSSend (handle, "<select name=\"Pull_down\" size=1>");
 HSSend (handle, "<option>Option1</option>");
 HSSend (handle, "<option>Option2</option>");
 HSSend (handle, "<option>Option3</option>");
 HSSend (handle, "<option>Option4</option>");
```

```
 HSSend (handle, "</select></p>");
 HSSend (handle, "<p>Enter some text here: <input type=text
size=20 maxlength=256 name=\"text_box\"></p>");
 HSSend (handle, "<p>Some more text here: <input type=text
size=20 maxlength=256 name=\"more_text\"></p>");
 HSSend (handle, "</form>");
 HSSend (handle, "</body>");
 HSSend (handle, "</html>");
} /* HCC --> File: normal.htm }} */
/* HCC --> File: normal.htm (from HTML) Post_my_post {{ */
void Post_my_post(unsigned long handle)
{
 /* add your implementation here: */
}
```

Post_my_post *function*

The completed `Post_my_post` function looks like this:

```
void Post_my_post(unsigned long handle)
{
 char Pull_down[8];
 char text_box[256];
 char more_text[256];
/* Use HSGetValue to retrieve values posted by the user */
 HSGetValue (handle, "Pull_down", Pull_down, 8);
 HSGetValue (handle, "text_box", text_box, 256);
 HSGetValue (handle, "more_text", more_text, 256);
/* At this point, the local variables Pull_down, text_box, and
more_text contain the values entered by the user via the web
browser. The application code can react accordingly.*/
}
```

Password and username authentication

The HTTP server supports password authentication. Users are prompted for a username and a password on pages designed to require that information for access. Applications use the `HSPageAuthenticate` routine to set the access status for each HTML page contained in the application.

HTTP server API functions

The following pages describe the HTTP server API functions.

HSCustomPage

Lets you customize your response page.

If you do not call this routine, the default "Not found" or "protected" page will be used.

You can call this function anywhere in your application.

Format

```
int HSCustomPage (int type, char *htmlbuffer);
```

Arguments

Argument	Description
type	One of the following types of response pages: `ACCESSDENIED` — after login fails for a protected URL `PAGENOTFOUND` — for an unknown URL
htmlbuffer	`NULL`-terminated string corresponding the HML text of the reply. The contents will be copied to an internal buffer on the HTTP server, so that you can safely reuse or discard the buffer after calling this function.

Return values

none

HSGetValue

This routine is used by applications that process HTML form data. The data returned from this routine is already converted from CGI format to its original format.

Format

```
int HSGetValue (unsigned long handle, char *namep,
       char* valuep,int maxlen);
```

Arguments

Argument	Description
handle	Network connection used for the request. This value is not user-modified, but must be attached to any calls to the server.
namep	Pointer to the name whose value is requested by the caller.
valuep	Pointer to the value of the specified name, if found.
maxlen	Maximum storage length pointed at by *valuep*.

Return values

Return value	Description
0 or greater	Success (This is the string length.)
-1	Failure

HSPageAccess

Controls Web page access via username and password.

Note: Deprecated function. Instead, the system access database should be populated by calling `NAsetSysAccess` (in `sysAccess.h`). If you use `HSPageAccess`, it will compile. However, system user account access must be made through the system access database API.

Format

```
void HSPageAccess(unsigned long handle, char* groupname,
        char* username, char* password);
```

Arguments

Argument	Description
handle	Network connection used for the request. This value is not user-modified, but must be attached to any calls to the server.
groupname	Pointer to the name used in the password dialog issued by the browser on behalf of the HTTP server prompting for a username and password. Limit 31 characters. If *groupname* is null (zero length), the page is accessible without a password.
username	Pointer to the username required for access to a page. Limit 31 characters.
password	Pointer to the password required for access to a page. Limit 31 characters.

Return values

none

HSPageAuthenticate

Controls Web page access via username and password.

Format

```
void HSPageAuthenticate(unsigned long handle,
        unsigned short groupname);
```

Arguments

Argument	Description
handle	Network connection used for the request. This value is not user-modified, but must be attached to any calls to the server.
groupname	Pointer to the name used in the password dialog issued by the browser on behalf of the HTTP server prompting for a username and password. Limit 31 characters. If *groupname* is null (zero length), the page is accessible without a password.

Return values

none

Usage notes

- This function should not be called directly. It should be called inside the `AppPreProcess` routine for each URL. The HTML-to-C Converter generates the calls.
- To assign multiple realms to a URL, use the + operator — for example, *Realm1 + Realm2*.
- The realm's username and password should be set in `HSInitSecurityTable`.

HSProperties

Modifies task properties of the HTTP server, such as stack size and priority.

Use this function only if you are familiar with the operating system characteristics that underlie the task properties. Usually, the default settings for the HTTP server are sufficient for most applications.

Format

```
void HSProperties (char* tnamep,int priority,int sysstack,
       int usrstack,int mode,int flags);
```

Arguments

Argument	Description
tnamep	Pointer to name of the HTTP server task.
priority	Server task priority. This value depends on the host operating system.
sysstack	Sze of the HTTP server supervisor stack. This sets the stack size on the server operating system.
usrstack	Size of the HTTP server user stack. This sets the user stack size.
mode	Additional task startup information for the HTTP server. This value depends on the host operating system.
flags	Additional task startup information for the HTTP server. This value is based on the host operating system.

Return values

none

HSRegisterSearchFunction

Supplies the HTTP server with the routine that matches browser requests with the content to be supplied by an application.

Format

```
void HSRegisterSearchFunction,(void (*app_search_URL)
       (unsigned long,char *URLp),void (*app_security_URL)
        (unsigned long char *URLp));
```

Arguments

Argument	Description
app_search_URL	Pointer to the application function to be called by the HTTP server for matching URLs.
app_security_URL	Pointer to the application function to be called by the HTTP server to set the access properties for the HTML page.

Return values

none

Usage notes

- See the example application in `src\examples\nahttp` for how this function is being called (in `root.c`.)
- Also, refer to `url.c` for the default implementation of `app_search_URL` and `app_security_URL`.

HSRemoteAddress

Enables the application to retrieve the IP address of the requesting browser.

Format

```
unsigned long HSRemoteAddress(unsigned long handle);
```

Arguments

Argument	Description
handle	Network connection used for the request. This value is not user-modified, but must be attached to any calls to the server.

Return values

IP address of the request browser

HSSend

Enables an application function to return HTML page content to a requesting browser. The HTML-to-C Converter calls this function in the routines generated for the user application.

Format

```
int HSSend(unsigned long handle,char *datap);
```

Arguments

Argument	Description
handle	Network connection used for the request. This value is not user-modified, but must be attached to any calls to the server.
datap	Pointer to a `NULL`-terminated string sent from the HTTP server back to the requesting browser.

Return values

Return value	Description
0 or greater	Success — the data was sent or queued to be sent
-1	Failure — an error prevented the data from being sent

HSSendBinary

Enables the return of binary data, such as images and Java applets, to a requesting browser.

Format

```
int HSSendBinary(unsigned long handle,char *datap,int len);
```

Arguments

Argument	Description
handle	Network connection used for the request. This value is not user-modified, but must be attached to any calls to the server.
datap	Pointer to binary data of size *len*, cast as a `char *`, which is sent by the HTTP server back to the requesting browser.
len	Size of the *datap* buffer.

Return values

Return value	Description
0 or greater	Success — the data was sent or queued to be sent
-1	Failure — an error prevented the data from being sent

HSSetRealm

Sets up an individual realm's name, username, and password.

Note: Deprecated function. Instead, the system access database should be populated by calling `NAsetSysAccess` (in `sysAccess.h`). If you use `HSSetRealm`, it will compile. However, system user account access must be made through the system access database API.

Format

```
int HSSetRealm(int realmno, char *realmname, char *username,
        char *password);
```

Arguments

Argument	Description
realmno	Integer specifying the particular realm as defined: 0 — Realm 1 1 — Realm 2 2 — Realm 3 3 — Realm 4 4 — Realm 5 5 — Realm 6 6 — Realm 7 7 — Realm 8
realmname	Pointer to the name used in the password dialog issued by the browser on behalf of the HTTP server prompting for a username and password.
username	Pointer to the username required for access to a page.
password	Pointer to the password required for access to a page.

Return values

Return value	Description
0	Success
-1	Failure

HSStartServer

Enables an application to start the HTTP server. The server runs on a task; the server must be called to start.

You must use `HSRegisterSearchFunction` before using `HSStartServer`.

Format

```
int HSStartServer (void);
```

Arguments

none

Return values

Return value	Description
0	Success — the server started successfully
-1	Failure — there was an error during server startup

MIME content functions

The following functions set the MIME type for the component being returned by the HTTP server. The functions are called from the authentication routine since that routine is called prior to the application search routine. For each Web component, a call to one of the functions listed is used to properly set the MIME type for the component.

For example, if the MIME content type is not set to `image/gif` for a `.gif` image, the requesting browser will not display the image properly. The HTML-to-C Converter generates the necessary call so that the HTTP server returns the information in the appropriate format.

Formats

```
void HSTypeHtml (long handle);
void HSTypeGif (long handle,);
void HSTypeApplet long handle, int length);
void HSTypeText (long handle);
void HSTypeJpeg (long handle, int length);
void HSTypePict (long handle, int length);
void HSTypeTiff (long handle, int length);
void HSTypePng (long handle, int length);
void HSTypeXbm (long handle, int length);
void HSTypeWav (long handle, int length);
void HSTypeAu (long handle, int length);
void HSTypeStaticApplet (long handle, int length);
void HSTypeStaticGif (long handle, int length);
void HSTypeStaticHtml (long handle);
void HSTypeStaticJpeg (long handle, int length);
void HSTypeStaticPict (long handle, int length);
void HSTypeStaticPng (long handle, int length);
void HSTypeStaticText (long handle);
void HSTypeStaticTiff (long handle, int length);
void HSTypeStaticWav (long handle, int length);
void HSTypeStaticAu (long handle, int length);
void HSTypeStaticXbm (long handle, int length);
```

Arguments

Argument	Description
handle	Network connection used for the request. This value is not user-modified, but must be attached to any calls to the server.
length	Length of the binary data being sent back. This value is placed in the `Content Length` value of the HTTP reply header.

Return values

none

Notes

The following MIME types are available (in order of the declarations):

- text / html
- image / gif
- application / octet-stream
- text / text
- image / jpeg
  image / pict
  image / tiff
  image / png
  image / x-xbitmap
- audio / wav
  audio / au

By default, caching is disabled. This means that every time you revisit a page, every object in the page will be sent from the server. To reduce network traffic for objects that never get changed, call `HSTypeStatic*`. For example, for `.gif` files, call `HSTypeStaticGif`.

HTTP server stub functions

The following section describes the HTTP server stub functions:

- `HSInitSecurityTable`
- `SearchURL`

These stub functions are in the `url.c` file.

HSInitSecurityTable

Called during the startup of the HTTP server to set up each realm's security information. The server supports a maximum of 8 realms.

This stub function is generated by the HTML-To-C Converter as an empty shell routine. Add calls to `HSSetRealm` to set up each realm's information.

Format

```
void HSInitSecurityTable(void);
```

Arguments

none

Return values

none

SearchURL

Called by the HTTP server when trying to locate a user application Web page.

Format

```
int SearchURL(unsigned long handle, char *URLp);
```

Arguments

Argument	Description
handle	Network connection used for the request. This value is not user-modified, but must be attached to any calls to the server.
URLp	Pointer to the client's requested URL.

Return values

Return value	Description
0	URL was not found
1	URL was found

LDAP Services API

CHAPTER 14

Overview

The Lightweight Directory Access Protocol (LDAP) uses a client-server architecture to provide a directory service that runs over TCP/IP port 389.

Use the LDAP services API to:

- Add, delete, and modify object attributes
- Add and delete obejcts
- Search for objects having a specified attribute

Include file

Using the LDAP services API requires the following header file:

```
ds_api.h
```

Summary of LDAP services API functions

Function	Description
`ds_add_attr`	Adds an attribute to a specified object.
`ds_delete_attr`	Deletes an object attribute from an object.
`ds_modify_attr`	Modifies an attribute of an object.
`ds_add_service`	Adds an object to the directory.
`ds_delete_service`	Deletes an object from the directory.
`ds_search_service`	Searches for objects having a specified attribute.
`ds_free_search_res`	Frees the allocated memory after a search.

LDAP services API functions

The following pages describe the LDAP services API functions.

Except where noted otherwise, these functions return `LDAP_SUCCESS` on successful completion, or pass through the error code from the calling software.

ds_add_attr

Adds an attribute to a specified object in the directory.

For example, you may want to add a name attribute like “Internet printer” for a printer object.

Format

```
int ds_add_attr (char *dn_user, char *pw,
    char *dn_obj, ds_attr_t *attr);
```

Arguments

Argument	Description
dn_user	Distinguished name of the user who has the authority to add the attribute.
pw	Password for the specified user.
dn_obj	Distinguished name of the device or service to which you are adding the attribute.
attr	Attribute to be added (type and values).

ds_delete_attr

Deletes an attribute from a specified object in the directory.

For example, you may want to delete the location attribute for a printer object.

Format

```
int ds_delete_attr(char *dn_user, char *pw,
    char *dn_obj, char *attr_type);
```

Arguments

Argument	Description
dn_user	Distinguished name of the user who has the authority to delete the attribute.
pw	Password for the specified user.
dn_obj	Distinguished name of the device or service for which you are deleting the attribute.
attr_type	Attribute type to be deleted.

ds_modify_attr

Modifies an attribute from a specified object in the directory.

For example, you may want to modify the description attribute for a printer object, such as "Printer temporarily out of order."

Format

```
int ds_modify_attr(char *dn_user, char *pw, char *dn_obj,
    ds_attr_t *attr);
```

Arguments

Argument	Description
dn_user	Distinguished name of the user who has the authority to modify the attribute.
pw	Password for the specified user.
dn_obj	Distinguished name of the device or service for which you are modifying the attribute.
attr	Attribute to be modified.

ds_add_service

Adds a device or service object to a directory.

Format

```
int ds_add_service (char *dn_user, char *pw,
    char *dn_obj, ds_attr_t **attr_list);
```

Arguments

Argument	Description
dn_user	The name of the user who has the authority to add the object.
pw	Password for the specified user.
dn_obj	Name of the device or service to be added.
attr_list	Attribute list for the object.

ds_delete_service

Deletes a device or service object from a directory.

Format

```
int ds_delete_service (char *dn_user, char *pw,
    char *dn_obj);
```

Arguments

Argument	Description
dn_user	Distinguished name of the user who has the authority to delete the object.
pw	Password for the specified user.
dn_obj	Distinguished name of the device or service to be deleted.

ds_search_service

Searches a device or service for a specified attribute and returns a list of objects.

The function allocates the memory for each of the obtained attributes and returns the pointer. Therefore, if the return value is not `NULL`, you must call `ds_free_search_res` to free the allocated memory.

Format

```
ds_search_list_t ds_search_service (char *dn_user, char *pw,
    char *attribute, char *value, char *base);
```

Arguments

Argument	Description
dn_user	Distinguished name of the user, or `NULL`.
pw	Password for the specified user.
attribute	Attribute name to be found.
value	Attribute value.
base	Distinguished name of the entry at which to start the search.

Return values

Pointer to a structure of type `ds_search_list_t`, defined as follows:

```
typedef struct  {
   int obj_num; /* Number of objects found */
   ds_search_res_t **objs; /* Pointer to address array of
structures  describing object matching the search criteria */
 } ds_search_list_t;
```

In this structure, `ds_search_res_t` is defined as follows:

```
typedef struct  {
    char *dn; /* Distinguished Name of object */
   ds_attr_t **attrs; /* Pointer to addresse array of attribute
structures */
 } ds_search_res_t;
```

ds_free_search_res

Frees the allocated memory from calling `ds_search_service`.

Use this call if the returned search is not `NULL`.

Format

```
int ds_free_search_res (ds_search_list_t *search_ptr);
```

Arguments

Argument	Description
search_ptr	Return value of `ds_search_service`.

Return values

Return value	Description
0	Success
-1	Failure

Management API

CHAPTER 15

Overview

Use the management API to:

- Create a list of all management variables on the system
- Protect management variables from concurrent access by multiple threads
- Read and write management variables
- Register functions that will be called when a variable is written to
- Check values that are written to variables (to make sure the values are within the valid ranges)

Note: For information on accessing management variables with Web pages, see Chapter 2 on the Advanced Web Server API.

Include file

The following header file is required for the management API:

```
man_api.h
```

Summary of management API functions

Function	Description
manAddVariableList	Adds a list of variable to the master list.
manAddVariableCallback	Registers a callback notification for a variable.
manDeleteSnmpRow	Deletes a row from an SNMP table.
manDeleteVariableCallback	De-registers a callback function for a management variable.
manDeleteVariableList	Deletes a list of variables from the master list.
manGetArray	Reads an array variable.
manGetChar	Reads a character variable.
manGetINT*n*	Reads a variable of the specified size — INT8, INT16, INT32, or INT64.
manGetOctetString	Reads an octet string variable.
manGetSnmpRow	Reads a row from an SNMP table.
manGetSnmpRowPos	Translates an SNMP index into a numeric index.
manGetUnknown	Reads a variable of an unknown type.
manGetVariableInfo	Gets information about a management variable.
manGetWORD*n*	Reads a variable of the specified size — WORD8, WORD16, WORD32, or WORD14.
manInsertSnmpRow	Inserts a row into an SNMP table.
manRegisterChangeFn	Registers a function to be called whenever a management variable changes.
manSetArray	Writes an array variable.
manSetChar	Writes a character variable.

Function	Description
manSetINT*n*	Writes a variable of the specified size — INT8, INT16, INT32, or INT64.
manSetOctetString	Writes an octet string variable.
manSetSnmpRow	Writes to a specified row in man SNMP table.
manSetUnknown	Writes a variable of an unknown type.
manSetWORD*n*	Writes a variable of the specified size — WORD8, WORD16, WORD32, or WORD14.
manUnregisterChangeFn	Unregisters a callback function.

Defining lists of variables

You must define a list of variables that will be accessed through the management API. These lists are defined as arrays of manVarType elements. Each element in a list defines a management variable. Each management variable must be given a unique identifier (NULL-terminated string).

You can create as many different lists of management variables as desired, and the lists can be any size. However, each idenitifer must be unique in all lists.

The manVarType data type is defined as follows:

```
typedef struct
{
    MAN_ID_TYPE id;
    void *varPointer;
    int isFunction;
    int size;
    int type;
    int *dimensions;
    int numberDimensions;
    MAN_SEMAPHORE *semaphores;
    int numberSemaphores;
    void *rangeFn;
    void *rangeInfo;
    manTableInfoType *tableInfo;
    void *callbackFn;
} manVarType;
```

The following table describes the fields in manVarType:

Field	Description
id	Must be loaded with a unique string that identifies the variable. MAN_ID_TYPE is defined as a pointer to a NULL-terminated character string.
varPointer	If *isFunction* is set, then *varPointer* should point to a function of type manAccessFunctionType; otherwise it should be set to NULL.
isFunction	Set to MAN_FN if *varPointer* contains a function pointer; otherwise, set to 0.
size	If *type* is set to MAN_UNKNOWN, then *size* must be set to the size of the variable in bytes.
type	Set to one of the type constants defined in man_api.h to indicate the data type of the variable.
dimensions	Set to NULL if the variable is a singleton. If the variable is an array, *dimensions* is an integer array that contains the variable's dimensions.
numberDimensions	Set to 0 if the variable is a singleton. If the variable is an array, *numberDimensions* must be set to the number of dimensions it has.
semaphores	Set to NULL if the variable does not have semaphores. If the variable is protected by semaphores, *semaphores* is an array of semaphores which the management module will lock before attempting to access the variable. The semaphores will be locked in the order they appear in the array, and unlocked in the reverse order. MAN_SEMAPHORE_TYPE is a pointer to a semaphore structure.
numberSemaphores	Set to 0 if *semaphores* is NULL. If *semaphores* is not NULL, then *numberSemaphores* must be loaded with the number of semaphores in *semaphores*.

Field	Description
rangeFn	Pointer to a function of type manTestFn which will be called when the application tries to write to the variable. This function is to verify that legal values are being written. `typedef MAN_error_type (*manTestFn) (manVarType *var, void *buffer)` where: ■ *var* is a pointer to the variable being accessed ■ *buffer* is pointer to value to be written to the variable If no verification function is to be used, set *rangeFn* to NULL.
rangeInfo	Can be used to store application-specific information about the range of legal values for the variable. Otherwise, set to NULL.
tableInfo	If *type* is MAN_SNMP_TABLE, then *tableInfo* is a pointer to a structure (see below) that has information about the table. Otherwise, set *tableInfo* to NULL.
callbackFn	For internal use only.; set to NULL.

manTableInfoType structure

The manTableInfoType is used to describe an SNMP table. When an SNMP table variable is created, the *tableInfo* field in its manVarType entry must point to one of these structures. Rows in the table will be represented as C structures. The manTableInfoType contains information that describes the fields inside the structure. Indexing information for the table is also stored in the structure.

The structure is defined as follows:

```
    typedef struct
{
        int numberFields;   /* count of fields in table*/
        int *fieldType;     /* type of each field*/
        int fieldSize;      /* size of each field*/
        void *indexFn;      /* index function */
        void *indexInfo;    /* info for index function */
    } manTableInfoType;
```

The following table describes the fields in `manTableInfoType`:

Field	Description
numberFields	Number of fields in the table.
fieldType	Pointer to an array where each element is set to a constant defined in `man_api.h` that describes the field's type.
fieldSize	Pointer to array where each element indicates the size of a table field in bytes.

indexFn	Pointer to a function (supplied by the application) to locate rows in the table by their index
	Function must be of type `manIndexFucntionType` defined as follows:
	`typedef int (*manIndexFunctionType)`
	`(void *index, void *row,`
	`void *indexInfo);`
	The function should return -1 if *row* is before the *index*, 0 if *row* is equal to *index*, or +1 if the *row* is after the *index*.

indexInfo	Application-dependent information passed to the function pointed to by *indexFn*.

Initialization

Before using the management API, applications must register their variable lists by calling the `manAddVariableList` function. Each call adds one list of variables to the master list of variables.

The following code fragment demonstrates how the management API could be initialized. In this example, you add two lists into the master list of variables.

```
manAddVariableList (list1,
    sizeof(list1)/sizeof(manVarType));

manAddVariableList(list2,
    sizeof(list2)/sizeof(manVarType));
```

When you add a variable list, the management API allocates memory for all of the variables defined for it (except ones to be serviced by application functions), and zereos out the buffers.

Reading and writing variables, array elements, and arrays

The following sections explains how to use the management API for read and write operations.

Reading and writing singleton values

To provide compile time type checking, the management API defines type-specific functions to read and write (get and set) management variables. The functions take the form:

- `MAN_`*`error_type`* `manGet`*`datatype`* `(MAN_ID_TYPE` *`id`*`,` *`datatype`* `*`*`buffer`*`, int *`*`indices`*`, MAN_TIMEOUT_TYPE` *`timeout`*`)`
- `MAN_`*`error_type`* `manSet`*`datatype`* `(MAN_ID_TYPE` *`id`*`,` *`datatype`* `*`*`value`*`, int *`*`indices`*`, MAN_TIMEOUT_TYPE` *`timeout`*`)`

where *`datatype`* is the data type of the variable being accessed. Functions are provided for reading and writing the following data types for singleton values:

- `char`
- `INT8 INT16 INT32 INT64`
- `WORD8 WORD16 WORD32 WORD64`

To determine the data type of a variable, use manGetVariableInfo.

For example, suppose a 32-bit integer variable had the unique identifier `MarketGarden`. The following code reads it into the `INT32` named `Arnhem`. If the variable is protected by a semaphore that is locked by another process, the code will wait indefinitely for it to be released.

```
IT32 Arnhem;
manGetInt32 ("MarketGarden", &Arnhem, NULL,
     MAN_TIMEOUT_FOREVER);
```

The following function call writes 0 to the variable:

```
manSetInt32 ("MarketGarden", 0, NULL, MAN_TIMEOUT_FOREVER);
```

Reading and writing array elements

You can use the same functions for reading singleton values to read and write individual elements of array variables. The *`indices`* argument specifies the coordinates of the element to access.

For example, a variable call `Leyte` is a 3-dimensional array of 16-bit words. The following code reads the element at coordinates (5, 2, 0):

```
WORD16 Gulf;
int indices[] = {5, 2, 0};
manGetWORD16 ("Layte", &Gulf, indices,
    MAN_TIMEOUT_FOREVER, NULL);
```

Reading and writing arrays

Use the `manGetArray`and `manSetArray` functions to read and write entire arrays. These functions support reading and writing only simple arrays, not structured arrays.

For example, amanagement variable called `Kursk` is a 5 x 10 array of 32-bit integers. The following code copies the contents of it into the buffer Citadel:

```
INT32 Citadel[5][10];
manGetArray ("Kursk", Citadel, sizeof (Citadel),
    MAN_TIMEOUT_FOREVER);
```

Accessing variables outside the management database

Use manAccessFunctionType to create functions to read and write variables that are not in the management database, but which will be accessed through the management API.

```
typedef MAN_error_type (*manAccessFunctionType)
    (manVarType *var, void *buffer, int *indices,
    int isWrite, MAN_TIMEOUT_TYPE timeout);
```

This function will be called after the API has locked any semaphores protecting the variable, and has confirmed that the application has permission to access the variable. The function should perform the operation and return MAN_SUCCESS or return one of the other error codes defined in man_api.h.

The following table describes the fields in manAccessFunctionType:

Field	Description
var	Pointer to list entry for the variable.
buffer	For read operations, pointer to the buffer to copy the variable to. For write operations, *buffer* contains the value to be written.
indices	Pointer to an array of subscripts for the elements to access. Set to NULL if the entire array is being read or written.
isWrite	Set to MAN_DO_READ or MAN_DO_WRITE.
timeout	Maximum time to wait for the variable to become accessible. See the section on "*Specifying timeouts*," below.

Accessing tables outside the management database

Use manTableAccessFunctionType to create functions to read, write, insert, and delete rows into table variables that are not in the management database, but which will be accessed through the management API.

```
typedef MAN_error_type (*manTableAccessFunctionType)
    (manVarType *var, void *buffer,
    manTableIndexType *index,
   manTableIndexType *newIndex, int operation,
   MAN_TIMEOUT_TYPE timeout);
```

This function will be called after the API has locked any semaphores protecting the variable, and has confirmed that the application has permission to access the variable. The function should perform the operation and return `MAN_SUCCESS` or return one of the other error codes defined in `man_api.h`.

The following table describes the fields in `manAccessFunctionType`:

Field	Description
var	Pointer to list entry for the variable.
buffer	For read (get) operations, pointer to the buffer to copy the variable to. For write (set) operations, *buffer* contains the value to be written.
index	Index information to determine which row in the table is being accessed.
newIndex	If *operation* is `MAN_WRITE`, then *newIndex* must contain the new index for the row.
operation	One of the following: ■ `MAN_DELETE` ■ `MAN_INSERT` ■ `MAN_READ` ■ `MAN_WRITE`
timeout	Maximum time to wait for the variable to become accessible. See the section on "*Specifying timeouts*," below.

manTableIndexType structure

Functions to access rows in tables take a parameter of type `manTableIndexType`. This type is used to store information about how to identify which row in a table will be operated on. The row can be identified by either a numeric index or by an SNMP index.

The `manTableIndexType` structure is defined as follows.

```
typedef struct
{
    int numericIndex;
    int wantExact;
    void *snmpIndex;
} manTableIndexType;
```

The following table describes the fields in `manTableIndexType`:

Field	Description
numericIndex	Zero-based number indicating the row position in the table. Thus, a *numericIndex* of 0 indicates the first row in the table should be accessed.
wantExact	Set only if an exact match to *snmpIndex* is acceptable; set to 0 to indicate the first row that exactly matches the index or comes after it.
snmpIndex	Stores SNMP indexing information. Application-dependent.

Specifying timeouts

In specifying the *timeout* parameter in the management API functions, you should generally specify the maximum number of ticks the application can wait for the variable to become accessible.

If the application can wait indefinitely for the variable to become accessible, specify `MAN_TIMEOUT_FOREVER`.

If you want the function to return immediately without waiting for the variable to become accessible, specify `MAN_TIMEOUT_NO_WAIT`.

Management API functions

The following pages describe the management API functions.

manAddVariableList

Adds a list of variables to the master list.

Format

```
MAN_error_type manAddVariableList (manVarType *varList,
       int numberVars);
```

Arguments

Argument	Description
varList	Pointer to an array that is a list of variables to add to the master list, or the definition of manVarType, see the section called "*Defining lists of variables,*" earlier in this chapter.
numberVars	Number of variables in the list.

Return values

Return value	Description
MAN_SUCCESS	Successfully added variables to the master list
MAN_DUPLICATE	One or more variables in the list have unique identifiers already in use in the master list
MAN_OUT_OF_MEMORY	Cannot allocate memory to expand the master list
MAN_NULL_POINTER	List specified was NULL
MAN_BAD_ID	Invalid ID field in one of the entries
MAN_BAD_DIMENSION_COUNT	Invalid dimension count in one of the entries
MAN_BAD_SEMAPHORE_COUNT	Invalid semaphonre count in one of the entries
MAN_BAD_VAR_POINTER	Invalid variable pointer in one of the entries
MAN_BAD_DIMENSION	Dimension in one of the entries is less than 1
MAN_BAD_SEMAPHORE	Invalid semaphore in one of the entries
MAN_BAD_SIZE	Entry of type MAN_UNKNOWN but size field < 1
MAN_BAD_TYPE	Invalid type field in one of the entries
MAN_BAD_TABLE_INFO	Invalid table information for type MAN_SNMP_TABLE

manAddVariableCallback

Registers a management variable for a callback notification.

Format

```
MAN_error_type manAddVariableCallback (MAN_ID_TYPE id,
int *indices, manCallbackFunctionType *fn, void *fnd);
```

Arguments

Argument	Description
id	Unique identifier of the variable to read.
indices	Pointer to an array of subscripts for the elements to access. If the variable is a singleton, *indices* should be NULL.
fn	Pointer to the callback function.
fnd	Pointer to the data structure to pass as a callback function parameter.

The manCallbackFunctionType structure is defined as follows:

```
typedef void (*manCallbackFunctionType) (MAN_ID_TYPE id,
        void *buf, int buflen, void *fnd);
```

Return values

Return value	Description
MAN_SUCCESS	Successfully registered the callback function
MAN_UNKNOWN_ID	No variable with the specified identifier.
MAN_NOT_ARRAY	Variable not an array, but *indices* not NULL
MAN_INVALID_SUBSCRIPT	One or more array subscripts out of bounds
MAN_MISSING_SUBSCRIPT	Variable is an array, but *indices* set to NULL

manDeleteSnmpRow

Deletes a row from an SNMP table.

Format

```
MAN_error_type manDeleteSnmpRow (MAN_ID_TYPE id,
      manTableIndexType index, MAN_TIMEOUT_TYPE timeout);
```

Arguments

Argument	Description
id	Unique indentifier of the variable to read.
index	Pointer to the structure with indexing information. If more than one row satisfies the *index*, only the first one encountered in the table is deleted.
timeout	Maximum time to wait for the variable to become accessible. See the section on "*Specifying timeouts,*" earlier in this chapter.

Return values

Return value	Description
MAN_SUCCESS	Successfully deleted the row
MAN_INCORRECT_TYPE	Variable is not an SNMP table
MAN_UNKNOWN_ID	No variable with the specified identifier
MAN_SEMAPHORE_LOCKED	Timeout expired before variable was ready
MAN_COMMUNICATIONS_FAILURE	Cannot communicate with external device where variable is located
MAN_INVALID_SUBSCRIPT	No row matches the specified index
MAN_NULL_POINTER	Row is set to NULL
MAN_BAD_TABLE_INFO	Invalid table information for type MAN_SNMP_TABLE
MAN_BAD_INDEX_FUNCTION	Index function not defined

manDeleteVariableCallback

Unregisters a callback function for a management variable.

Format

```
MAN_error_type manDeleteVariableCallback(MAN_ID_TYPE id,
        int *indices, manCallbackFunctionType fn);
```

Arguments

Argument	Description
id	Unique identifier of the variable to callback.
indices	Pointer to an array of subscripts for the elements to access. If the variable is a singleton, *indices* should be `NULL`.
fn	Pointer to the callback function previously registered. For the type definition of `manCallbackFunctionType`, see the description of the `manAddVariableCallback` function.

Return values

Return value	Description
`MAN_SUCCESS`	Successfully unregistered the callback function
`MAN_UNKNOWN_ID`	No variable with the specified identifier
`MAN_UNKNOWN_FN`	No callback function currently registered with the specified identifier
`MAN_INVALID_SUBSCRIPT`	No row matches the specified index
`MAN_NOT_AN_ARRAY`	Variable is not an array
`MAN_MISSING_SUBSCRIPTS`	Variable is an array but *indices* set to `NULL`

manDeleteVariableList

Deletes a list of variables from the master list.

Format

```
MAN_error_type manDeleteVariableList (manVarType *varList);
```

Arguments

Argument	Description
varList	Pointer to an array that is a list of variables to delete. For the definition of `manVarType`, see the section called "*Defining lists of variables*," earlier in this chapter.

Return values

Return value	Description
MAN_SUCCESS	Successfully deleted variables from the master list
MAN_UNKNOWN_LIST	List was not registered with the management API
MAN_OUT_OF_MEMORY	Cannot allocate memory to rebuild the master list hash table
MAN_NULL_POINTER	*varList* passed as NULL
MAN_UNKNOWN_FN	Error deleting a callback function

manGetArray

Reads an array variable.

Format

```
MAN_error_type manGetArray (MAN_ID_TYPE id, void *buffer,
       int size, MAN_TIMEOUT_TYPE timeout);
```

Arguments

Argument	Description
id	Unique indentifier of the variable to read.
buffer	Buffer for copying the array.
size	Size of the buffer. Must be large enough for all of the array variable.
timeout	Maximum time to wait for the variable to become accessible. See the section on "*Specifying timeouts*," earlier in this chapter.

Return values

Return value	Description
MAN_SUCCESS	Successfully read the array
MAN_UNKNOWN_ID	No variable with the specified identifier
MAN_SEMAPHORE_LOCKED	Timeout expired before variable was ready
MAN_COMMUNICATIONS_FAILURE	Cannot communicate with external device where variable is located
MAN_NOT_AN_ARRAY	Variable not an array, but *indices* not NULL
MAN_BUFFER_WRONG_SIZE	Buffer is not the same size as the array variable
MAN_STRUCTURED_ARRAY	Function not supported on structured arrays, like octet strings, or SNMP tables

manGetChar

Reads a character variable.

Format

```
MAN_error_type manGetChar (MAN_ID_TYPE id,
        MAN_CHAR_TYPE *buffer, int indices,
        MAN_TIMEOUT_TYPE timeout);
```

Arguments

Argument	Description
id	Unique indentifier of the variable to read.
buffer	Buffer for copying the variable.
indices	Pointer to an array of subscripts for the elements to access. If the variable is a singleton, *indices* should be NULL.
timeout	Maximum time to wait for the variable to become accessible. See the section on "*Specifying timeouts*," earlier in this chapter.

Return values

Return value	Description
MAN_SUCCESS	Successfully read the variable
MAN_INCORRECT_TYPE	Variable is not a MAN_CHAR type
MAN_UNKNOWN_ID	No variable with the specified identifier
MAN_SEMAPHORE_LOCKED	Timeout expired before variable was ready
MAN_COMMUNICATIONS_FAILURE	Cannot communicate with external device where variable is located
MAN_NOT_AN_ARRAY	Variable not an array, but *indices* not NULL
MAN_INVALID_SUBSCRIPT	One or more array subscripts out of bounds
MAN_MISSING_SUBSCRIPTS	Variable is an array but *indices* set to NULL

manGetINT*n*

Reads an integer variable of the specified size — `INT8`, `INT16`, `INT32`, or `INT64`.

Format

```
MAN_error_type manGetINT8 (MAN_ID_TYPE id, INT8 *buffer,
        int indices, MAN_TIMEOUT_TYPE timeout);

MAN_error_type manGetINT16 (MAN_ID_TYPE id, INT16 *buffer,
        int indices, MAN_TIMEOUT_TYPE timeout);

MAN_error_type manGetINT32 (MAN_ID_TYPE id, INT32 *buffer,
        int indices, MAN_TIMEOUT_TYPE timeout);

MAN_error_type manGetINT64 (MAN_ID_TYPE id, INT64 *buffer,
        int indices, MAN_TIMEOUT_TYPE timeout);
```

Arguments

Argument	Description
id	Unique indentifier of the variable to read.
buffer	Buffer for copying the variable.
indices	Pointer to an array of subscripts for the elements to access. If the variable is a singleton, *indices* should be `NULL`.
timeout	Maximum time to wait for the variable to become accessible. See the section on "*Specifying timeouts*," earlier in this chapter.

Return values

Return value	Description
MAN_SUCCESS	Successfully read the variable
MAN_INCORRECT_TYPE	Variable is not INT8, INT16, INT32, or INT64.
MAN_UNKNOWN_ID	No variable with the specified identifier
MAN_SEMAPHORE_LOCKED	Timeout expired before variable was ready
MAN_COMMUNICATIONS_FAILURE	Cannot communicate with external device where variable is located
MAN_NOT_AN_ARRAY	Variable not an array, but *indices* not NULL
MAN_INVALID_SUBSCRIPT	One or more array subscripts out of bounds
MAN_MISSING_SUBSCRIPTS	Variable is an array but *indices* set to NULL

manGetOctetString

Reads an SNMP octet string variable.

Format

```
MAN_error_type manGetOctetString (MAN_ID_TYPE id,
        MAN_OCTET_STRING_TYPE *string, int indices,
        MAN_TIMEOUT_TYPE timeout);
```

Arguments

Argument	Description
id	Unique indentifier of the variable to read.
string	Pointer to the buffer for copying the variable.
indices	Pointer to an array of subscripts for the elements to access. If the variable is a singleton, *indices* should be NULL.
timeout	Maximum time to wait for the variable to become accessible. See the section on "*Specifying timeouts*," earlier in this chapter.

The MAN_OCTET_STRING_TYPE structure is defined as follows:

```
typedef struct
{
     unsigned char *buffer;
     int length;
     int maxLength;
} MAN_OCTET_STRING_TYPE;
```

where:

buffer	Pointer to the buffer that contains the octet string.
length	The number of bytes of data in the buffer (starting from byte 0).
maxLength	The length of the buffer in bytes.

The management API allocates buffers for octet strings in increments of 256 bytes. So, if the current buffer is 256 bytes long, and a new value is written that is 700 bytes long, then the current buffer will be released, and a new one, 768 bytes long, will be allocated. Writing a shorter value does not cause the buffer to be freed and reallocated.

Return values

Return values	Description
`MAN_SUCCESS`	Successfully read the variable
`MAN_INCORRECT_TYPE`	Variable is not an octet string
`MAN_UNKNOWN_ID`	No variable with the specified identifier
`MAN_SEMAPHORE_LOCKED`	Timeout expired before variable was ready
`MAN_COMMUNICATIONS_FAILURE`	Cannot communicate with external device where variable is located
`MAN_NOT_AN_ARRAY`	Variable not an array, but *`indices`* not `NULL`
`MAN_INVALID_SUBSCRIPT`	One or more array subscripts out of bounds
`MAN_MISSING_SUBSCRIPTS`	Variable is an array but *`indices`* set to `NULL`
`MAN_NULL_POINTER`	*`string`* or *`string`* -> *`buffer`* is `NULL` If string is not , *`string`* -> *`length`* will be set to the required buffer length.
`MAN_BUFFER_WRONG_SIZE`	*`string`* -> *`maxLength`* is shorter than the length of the buffer *`string`* -> *`length`* will be set to the required buffer length.

manGetSnmpRow

Reads a row from an SNMP table.

Format

```
MAN_error_type manGetSnmpRow (MAN_ID_TYPE id,
        manTableIndexType index, void *row,
        MAN_TIMEOUT_TYPE timeout);
```

Arguments

Argument	Description
id	Unique indentifier of the variable to access.
index	Pointer to the structure with indexing information. If more than one row satisfies the *index*, only the first one encountered in the table is read.
row	Pointer to a buffer where the row will be copied.
timeout	Maximum time to wait for the variable to become accessible. See the section on “*Specifying timeouts*,” earlier in this chapter.

Return values

Return value	Description
MAN_SUCCESS	Successfully read the row
MAN_BUFFER_WRONG_SIZE	One or more octet string buffers are too small
MAN_UNKNOWN_ID	No variable with the specified identifier
MAN_SEMAPHORE_LOCKED	Timeout expired before variable was ready
MAN_COMMUNICATIONS_FAILURE	Cannot communicate with external device where variable is located
MAN_INVALID_SUBSCRIPT	No row matches the specified index
MAN_NULL_POINTER	*row* is set to NULL
MAN_BAD_TABLE_INFO	Invalid table information for type MAN_SNMP_TABLE
MAN_BAD_INDEX_FUNCTION	Index function not defined

manGetSnmpRowPos

Translates an SNMP index into a numeric index.

Format

```
MAN_error_type manGetSnmpRowPos (MAN_ID_TYPE id,
        manTableIndexType index, int *pos,
        MAN_TIMEOUT_TYPE timeout);
```

Arguments

Argument	Description
id	Unique indentifier of the variable to access.
index	Pointer to the structure with indexing information. If more than one row satisfies the *index*, the position of the first one is returned.
pos	Pointer to the integer that will be loaded with the row's numeric position in the table.
timeout	Maximum time to wait for the variable to become accessible. See the section on "*Specifying timeouts*," earlier in this chapter.

Return values

Return value	Description
MAN_SUCCESS	Successfully read the position
MAN_UNKNOWN_ID	No variable with the specified identifier
MAN_SEMAPHORE_LOCKED	Timeout expired before variable was ready
MAN_COMMUNICATIONS_FAILURE	Cannot communicate with external device where variable is located
MAN_INVALID_SUBSCRIPT	No row matches the specified index
MAN_NULL_POINTER	*pos* is set to NULL

manGetUnknown

Reads a management variable of an unknown type.

Format

```
MAN_error_type manGetUknown (MAN_ID_TYPE id,
        void *buffer,int size, int indices,
        MAN_TIMEOUT_TYPE timeout);
```

Arguments

Argument	Description
id	Unique indentifier of the variable to access.
buffer	Buffer for copying the variable.
size	Size of the buffer. Must be large enough for all of the array variable.
indices	Pointer to an array of subscripts for the elements to access. If the variable is a singleton, *indices* should be NULL.
timeout	Maximum time to wait for the variable to become accessible. See the section on "*Specifying timeouts*," earlier in this chapter.

Return values

Return value	Description
MAN_SUCCESS	Successfully read the variable
MAN_INCORRECT_TYPE	Variable is not a MAN_UNKNOWN type
MAN_UNKNOWN_ID	No variable with the specified identifier
MAN_SEMAPHORE_LOCKED	Timeout expired before variable was ready
MAN_COMMUNICATIONS_FAILURE	Cannot communicate with external device where variable is located
MAN_NOT_AN_ARRAY	Variable not an array, but *indices* not NULL
MAN_INVALID_SUBSCRIPT	One or more array subscripts out of bounds
MAN_MISSING_SUBSCRIPTS	Variable is an array but *indices* set to NULL

manGetVariableInfo

Gets information about a management variable.

Format

```
MAN_error_type manGetVariableInfo (MAN_ID_TYPE id,
        int *variableType, int *size, int *dimensions,
        int *numberDimensions, WORD32 *access);
```

Arguments

Argument	Description
id	Unique indentifier of the variable to access.
variableType	On return, indicates the type of variable.
size	If the variable is an array, on return *size* will be loaded with the length of one element in bytes; if the variable is a singleton, *size* will be set to its size in bytes.
dimensions	Pointer to an integer array that, on return, will be loaded with variable's dimensions. If you are not interested in the dimensions, set to NULL.
numberDimensions	Must be set to the size of the *dimensions* array or to 0 (zero) if *dimensions* is *NULL*. On return, *numberDimensions* will be set to the number of dimensions the variable has, or to 0 if the variable is not an array.
access	Set by the function to indicate the type of access allowed by the variable: ■ MAN_READ ■ MAN_WRITE ■ MAN_READ | MAN_WRITE

Return values

Return value	Description
MAN_SUCCESS	Successfully read the variable
MAN_ARRAY_TOO_SMALL	*dimensions* array too small
MAN_UNKNOWN_ID	No variable with the specified identifier

manGetWORD*n*

Reads a word variable of the specified size — `WORD8`, `WORD16`, `WORD32`, or `WORD14`.

Format

```
MAN_error_type manGetWORD8 (MAN_ID_TYPE id, WORD8 *buffer,
        int indices, MAN_TIMEOUT_TYPE timeout);

MAN_error_type manGetWORD16 (MAN_ID_TYPE id, WORD16 *buffer,
        int indices, MAN_TIMEOUT_TYPE timeout);

MAN_error_type manGetWORD32 (MAN_ID_TYPE id, WORD32 *buffer,
        int indices, MAN_TIMEOUT_TYPE timeout);

MAN_error_type manGetWORD64 (MAN_ID_TYPE id, WORD64 *buffer,
        int indices, MAN_TIMEOUT_TYPE timeout);
```

Arguments

Argument	Description
id	Unique indentifier of the variable to access.
buffer	Buffer for copying the variable.
indices	Pointer to an array of subscripts for the elements to access. If the variable is a singleton, *indices* should be `NULL`.
timeout	Maximum time to wait for the variable to become accessible. See the section on "*Specifying timeouts*," earlier in this chapter.

Return values

Return value	Description
MAN_SUCCESS	Successfully read the variable
MAN_INCORRECT_TYPE	Variable is not WORD8, WORD16, WORD32, or WORD14
MAN_UNKNOWN_ID	No variable with the specified identifier
MAN_SEMAPHORE_LOCKED	Timeout expired before variable was ready
MAN_COMMUNICATIONS_FAILURE	Cannot communicate with external device where variable is located
MAN_NOT_AN_ARRAY	Variable not an array, but *indices* not NULL
MAN_INVALID_SUBSCRIPT	One or more array subscripts out of bounds
MAN_MISSING_SUBSCRIPTS	Variable is an array but *indices* set to NULL

manInsertSnmpRow

Inserts a row into an SNMP table.

Format

```
MAN_ERROR_TYPE manInsertSnmpRow (MAN_ID_TYPE id,
        manTableIndexType index, void *row,
        MAN_TIMEOUT_TYPE timeout);
```

Arguments

Argument	Description
id	Unique indentifier of the variable to access.
index	Pointer to the structure with indexing information. The row will be inserted before the first row≥ *index*.
row	Pointer to the row to be inserted.
timeout	Maximum time to wait for the variable to become accessible. See the section on "*Specifying timeouts*," earlier in this chapter.

Return values

Return value	Description
MAN_SUCCESS	Successfully inserted the row
MAN_INCORRECT_TYPE	Variable is not a table
MAN_UNKNOWN_ID	No variable with the specified identifier
MAN_SEMAPHORE_LOCKED	Timeout expired before variable was ready
MAN_COMMUNICATIONS_FAILURE	Cannot communicate with external device where variable is located
MAN_NULL_POINTER	Row is set to NULL
MAN_OUT_OF_MEMORY	Cannot allocate memory for the insert
MAN_BAD_TABLE_INFO	Invalid table information for type MAN_SNMP_TABLE
MAN_BAD_INDEX_FUNCTION	Index function not defined
MAN_DUPLICATE	One or more variables in the list have unique identifiers already in use in the master list

manRegisterChangeFn

Registers a function that is to be called whenever a management variable changes.

Format

```
MAN_error_type manRegisterChangeFunction
        (manVariableChangeFn fn);
```

Arguments

Argument	Description
fn	Pointer to the function to call when any variable is modified.

The `manVariableChangeFnType` structure is defined as follows:

```
typedef void (*manVariableChangeFnType) (manVarType *var,
        void *buffer, int *indices, int operation);
```

For the definition of `manVarType`, see the section called "*Defining lists of variables*," earlier in this chapter.

Return values

Return value	Description
MAN_SUCCESS	Successfully registered the callback function
MAN_ALREADY_REGISTERED	Another callback function was previously registered

manSetArray

Writes an array variable.

Format

```
MAN_error_type manSetArray (MAN_ID_TYPE id, void *buffer,
      int size, MAN_TIMEOUT_TYPE timeout);
```

Arguments

Argument	Description
id	Unique indentifier of the variable to access.
buffer	Buffer containing the data to write into the variable.
size	Size of the buffer. Must be large enough for all of the array variable.
timeout	Maximum time to wait for the variable to become accessible. See the section on "*Specifying timeouts*," earlier in this chapter.

Return values

Return value	Description
MAN_SUCCESS	Successfully wrote the variable
MAN_UNKNOWN_ID	No variable with the specified identifier
MAN_SEMAPHORE_LOCKED	Timeout expired before variable was ready
MAN_COMMUNICATIONS_FAILURE	Cannot communicate with external device where variable is located
MAN_NOT_AN_ARRAY	Variable is not an array
MAN_BUFFER_WRONG_SIZE	Buffer is not the same size as the array variable
MAN_STRUCTURED_ARRAY	Function not supported on structured arrays, octet strings, or SNMP tables
MAN_ILLEGAL_VALUE	Array contains illegal values for management variables
MAN_BAD_INDEX_FUNCTION	Index function not defined

manSetChar

Writes a character variable.

Format

```
MAN_error_type manSetChar (MAN_ID_TYPE id,
       MAN_CHAR_TYPE *newValue, int indices
       MAN_TIMEOUT_TYPE timeout);
```

Arguments

Argument	Description
id	Unique indentifier of the variable to access.
newValue	Value to write into the variable.
indices	Pointer to an array of subscripts for the elements to access. If the variable is a singleton, *indices* should be NULL.
timeout	Maximum time to wait for the variable to become accessible. See the section on "*Specifying timeouts*," earlier in this chapter.

Return values

Return value	Description
MAN_SUCCESS	Successfully wrote the variable
MAN_UNKNOWN_ID	No variable with the specified identifier
MAN_SEMAPHORE_LOCKED	Timeout expired before variable was ready
MAN_COMMUNICATIONS_FAILURE	Cannot communicate with external device where variable is located
MAN_NOT_AN_ARRAY	Variable is not an array, but *indices* not NULL
MAN_INCORRECT_TYPE	Variable is not a MAN_CHAR_TYPE
MAN_INVALID_SUBSCRIPT	One or more array subscripts out of bounds
MAN_MISSING_SUBSCRIPTS	Variable is an array but *indices* set to NULL
MAN_ILLEGAL_VALUE	Value specified is illegal for management variables

manSetINT*n*

Writes an integer variable of the specified size — `INT8`, `INT16`, `INT32`, or `INT64`.

Format

```
MAN_error_type manSetINT8 (MAN_ID_TYPE id, INT8 newValue,
        int indices, MAN_TIMEOUT_TYPE timeout);

MAN_error_type manSetINT16 (MAN_ID_TYPE id, INT16 newValue,
        int indices, MAN_TIMEOUT_TYPE timeout);

MAN_error_type manSetINT32 (MAN_ID_TYPE id, INT32 newValue,
        int indices, MAN_TIMEOUT_TYPE timeout);

MAN_error_type manSetINT64 (MAN_ID_TYPE id, INT64 newValue,
        int indices, MAN_TIMEOUT_TYPE timeout);
```

Arguments

Argument	Description
id	Unique indentifier of the variable to access.
newValue	Value to write into the variable.
indices	Pointer to an array of subscripts for the elements to access. If the variable is a singleton, *indices* should be `NULL`.
timeout	Maximum time to wait for the variable to become accessible. See the section on "*Specifying timeouts*," earlier in this chapter.

Return values

Return value	Description
MAN_SUCCESS	Successfully wrote the variable
MAN_UNKNOWN_ID	No variable with the specified identifier
MAN_SEMAPHORE_LOCKED	Timeout expired before variable was ready
MAN_COMMUNICATIONS_FAILURE	Cannot communicate with external device where variable is located
MAN_NOT_AN_ARRAY	Variable not an array, but *indices* not NULL
MAN_INCORRECT_TYPE	Variable is not INT8, INT16, INT32, or INT64
MAN_INVALID_SUBSCRIPT	One or more array subscripts out of bounds
MAN_MISSING_SUBSCRIPTS	Variable is an array but *indices* set to NULL
MAN_ILLEGAL_VALUE	Value specified is illegal for management variables

manSetOctetString

Writes an octet string variable.

Format

```
MAN_error_type manSetOctetString (MAN_ID_TYPE id,
        MAN_OCTET_STRING_TYPE *string, int indices,
        MAN_TIMEOUT_TYPE timeout);
```

Arguments

Argument	Description
id	Unique indentifier of the variable to access.
string	Pointer to the the string to write into the variable. For the definition of the `MAN_OCTET_STRING_TYPE` type, see the description of `manGetOctetString`.
indices	Pointer to an array of subscripts for the elements to access. If the variable is a singleton, *indices* should be `NULL`.
timeout	Maximum time to wait for the variable to become accessible. See the section on "*Specifying timeouts*," earlier in this chapter.

Return values

Return value	Description
MAN_SUCCESS	Successfully wrote the variable
MAN_INCORRECT_TYPE	Variable is not an octet string
MAN_UNKNOWN_ID	No variable with the specified identifier
MAN_SEMAPHORE_LOCKED	Timeout expired before variable was ready
MAN_COMMUNICATIONS_FAILURE	Cannot communicate with external device where variable is located
MAN_NOT_AN_ARRAY	Variable is not an array, but *indices* not NULL
MAN_INVALID_SUBSCRIPT	One or more array subscripts out of bounds
MAN_MISSING_SUBSCRIPTS	Variable is an array but *indices* set to NULL
MAN_ILLEGAL_VALUE	Value specified is illegal for this variable
MAN_NULL_POINTER	*string* or *string*->*buffer* is NULL

manSetSnmpRow

Writes to a specified row in an SNMP table.

Format

```
MAN_error_type manSetSnmpRow (MAN_ID_TYPE id,
        manTableIndexType index, manTableIndexType newIndex,
        void *value, MAN_TIMEOUT_TYPE timeout);
```

Arguments

Argument	Description
id	Unique indentifier of the variable to access.
index	Pointer to indexing information for the row to be set.
newIndex	Pointer to indexing information for the row's new position.
value	Pointer to a buffer that contains the new value for the row.
timeout	Maximum time to wait for the variable to become accessible. See the section on "*Specifying timeouts*," earlier in this chapter.

Return values

Return value	Description
MAN_SUCCESS	Successfully wrote to the row
MAN_INCORRECT_TYPE	Variable is not a table
MAN_UNKNOWN_ID	No variable with the specified identifier
MAN_SEMAPHORE_LOCKED	Timeout expired before variable was ready
MAN_COMMUNICATIONS_FAILURE	Cannot communicate with external device where variable is located
MAN_INVALID_SUBSCRIPT	No row matches the specified index
MAN_NULL_POINTER	*value* is set to NULL
MAN_OUT_OF_MEMORY	Cannot allocate memory

Return value	Description
MAN_BAD_TABLE_INFO	Invalid table information for type MAN_SNMP_TABLE
MAN_DUPLICATE	One or more variables in the list have unique identifiers already in use in the master list
MAN_BAD_INDEX_FUNCTION	Index function not defined

manSetUnknown

Writes a management variable of an unknown type.

Format

```
MAN_error_type manSetUnknown (MAN_ID_TYPE id,
        void *newValue, int size, int indices,
        MAN_TIMEOUT_TYPE timeout);
```

Arguments

Argument	Description
id	Unique indentifier of the variable to access.
newValue	Value to write into the variable.
size	Size of the buffer. Must be large enough for all of the array variable.
indices	Pointer to an array of subscripts for the elements to access. If the variable is a singleton, *indices* should be `NULL`.
timeout	Maximum time to wait for the variable to become accessible. See the section on "*Specifying timeouts*," earlier in this chapter.

Return values

Return value	Description
MAN_SUCCESS	Successfully wrote the variable
MAN_UNKNOWN_ID	No variable with the specified identifier
MAN_SEMAPHORE_LOCKED	Timeout expired before variable was ready
MAN_COMMUNICATIONS_FAILURE	Cannot communicate with external device where variable is located
MAN_NOT_AN_ARRAY	Variable is not an array, but *indices* not NULL
MAN_INCORRECT_TYPE	Variable is not a MAN_CHAR_TYPE
MAN_INVALID_SUBSCRIPT	One or more array subscripts out of bounds
MAN_MISSING_SUBSCRIPTS	Variable is an array but *indices* set to NULL
MAN_ILLEGAL_VALUE	Value specified is illegal for management variables

manSetWORD*n*

Writes a word variable of the specified size — `WORD8`, `WORD16`, `WORD32`, or `WORD64`.

Format

```
MAN_error_type manSetWORD8 (MAN_ID_TYPE id, WORD8 newValue,
        int indices, MAN_TIMEOUT_TYPE timeout);

MAN_error_type manSetWORD16 (MAN_ID_TYPE id, WORD16 newValue,
        int indices, MAN_TIMEOUT_TYPE timeout);

MAN_error_type manSetWORD32 (MAN_ID_TYPE id, WORD32 newValue,
        int indices, MAN_TIMEOUT_TYPE timeout);

MAN_error_type manSetWORD64 (MAN_ID_TYPE id, WORD64 newValue,
        int indices, MAN_TIMEOUT_TYPE timeout);
```

Arguments

Argument	Description
id	Unique indentifier of the variable to write.
newValue	Value to write into the variable.
indices	Pointer to an array of subscripts for the elements to access. If the variable is a singleton, *indices* should be `NULL`.
timeout	Maximum time to wait for the variable to become accessible. See the section on "*Specifying timeouts*," earlier in this chapter.

Return values

Return value	Description
MAN_SUCCESS	Successfully wrote the variable
MAN_UNKNOWN_ID	No variable with the specified identifier
MAN_SEMAPHORE_LOCKED	Timeout expired before variable was ready
MAN_COMMUNICATIONS_FAILURE	Cannot communicate with external device where variable is located
MAN_NOT_AN_ARRAY	Variable not an array, but *indices* not NULL
MAN_INCORRECT_TYPE	Variable is not WORD8, WORD16, WORD32, or WORD64
MAN_INVALID_SUBSCRIPT	One or more array subscripts out of bounds
MAN_MISSING_SUBSCRIPTS	Variable is an array but *indices* set to NULL
MAN_ILLEGAL_VALUE	Value specified is illegal for management variables

manUnregisterChangeFn

Unregisters a callback function.

Format

```
MAN_error_type manUnregisterChange Fn(void)
```

Arguments

none

Return values

Return value	Description
MAN_SUCCESS	Successfully unregistered the callback function
MAN_NOT_REGISTERED	Callback function was not registered

General-Purpose and System Access API

CHAPTER 16

Overview

The general-purpose API provides a group of convenience routines (for example, to convert the format of an IP addresses).

The system access API provides a way to set and get account security information. These function calls supersede some functions used in the HTTP server API.

Include file

Using the general-purpose API requires the following header file:

```
narmapi.h
```

Using the system access API requires the following header file:

```
sysAccess.h
```

Summary of general-purpose and system access API functions

Function	Description
NAInet_addr	Converts an IP address from a string into an unsigned long.
NATotalTicks	Returns the total number of ticks that have occurred since system startup.
NADeltaTicks	Calculates the number of ticks that have elapsed between the specified time and the current time.
NAInet_toa	Converts an IP address from an unsigned long into dotted notation (*a.b.c.d*).
System access (account security)	
NAgetSysAccess	Retrieves the password, account privileges, and IP address for an account.
NAsetSysAccess	Creates or removes records in the security access database.

General-purpose API functions

The following pages describe the general-purpose API functions.

NAInet_addr

Converts an IP address from a string into an unsigned long, in network byte order.

Format

```
unsigned long NAInet_addr (char *addrp);
```

Arguments

Argument	Description
addrp	Pointer to a NULL-terminated string representing an IP address (for example, 7.92.186.198).

Return values

Return value	Description
0	Invalid IP address
otherwise	Unsigned long representation of the IP address defined by the *addrp* argument

NATotalTicks

Returns the total number of ticks that have occurred since system startup.

Format

```
void NATotalTicks (unsigned long *hip, unsigned long *lop);
```

Arguments

Argument	Description
hip	Destination of high order word of tick clock.
lop	Destination of low order word of tick clock.

Return values

none

NADeltaTicks

Calculates the number of ticks that have elapsed between the specified time and the current time.

If this interval is too large, the results are inaccurate.

Format

```
unsigned long NADeltaTicks (unsigned long hiord,
        unsigned long loord);
```

Arguments

Argument	Description
hiord	High order word of start time.
loord	Low order word of start time.

Return values

Elapsed time in ticks

NAInet_toa

Converts an IP address from an unsigned long into dotted notation (*a.b.c.d*).

Format

```
void NAInet_toa (unsigned long ulipaddr, char *dest);
```

Arguments

Argument	Description
ulipaddr	Long format of the IP address to convert.
dest	Pointer to the destination of the character representation. This argument must be 16 or more characters long (15 characters plus the NULL terminator).

Return values

none

System access API functions

The following pages describe the system access API functions.

NAgetSysAccess

Retrieves the account password, privileges, and optional IP address filter.

Format

```
unsigned int NAgetSysAccess (char *user, char *password,
        unsigned int *ipAddr);
```

Arguments

Argument	Description
user	Pointer to the username of the account.
password	Pointer to a string buffer where the password is written. Minimum size of the buffer is `NASYSACC_STRLEN_PASSWORD + 1`. (`NASYSACC_STRLEN_PASSWORD` is defined in the `sysAccess.h` file.)
ipAddr	Pointer to the optional IP address filter. Can be set to `NULL` if your are not interested in this data.

Return values

The return value is a bit array containing any combination of the following access levels:

Return value	Description
NASYSACC_LEVEL_R	Account can be used to log in and read data
NASYSACC_LEVEL_RW	Account can be used to log in and read and write data
NASYSACC_LEVEL_HTTP_R*n*	Account can access Web pages in the indicated security realm 1–8 For example, NASYSACC_LEVEL_HTTP_R1 indicates the account can access Web pages in security realm 1.
NASYSACC_LEVEL_GATEWAY	Account can be used for the secure gateway
NASYSACC_LEVEL_SNMP_R	Account defines the SNMP public community string
NASYSACC_LEVEL_SNMP_RW	Account defines the SNMP private community string
NASYSACC_LEVEL_ROOT	Account has full (root) privileges
0	Account has no privileges, or the password is NULL

NAsetSysAccess

Creates or removes records in the security access database. The records include an account name, password, account privileges, and optional IP address.

The maximum number of accounts depends on NASYSACC_MAX_ACCOUNTS (defined in the sysAccess.h file). Default 12. (This value can be changed and recompiled, if necessary.)

Format

```
int NAsetSysAccess (unsigned int op, char *user,
       char *password, unsigned int level, char *ipAddr);
```

Arguments

Argument	Description
op	The type of operation used: ■ NASYSACC_ADD — creates the records in the database ■ NASYSACC_DEL — removes the account that matches the given username and password
user	Pointer to a string representing the username for the acccount. Maxium length is NASYSACC_STRLEN_USERNAME (defined in sysAccess.h) not including the string termination. Default 32. Note that changing this value may corrupt NVRAM.
password	Pointer to a string representing the password associated with *user*. Maxium length is NASYSACC_STRLEN_PASSWORD (defined in sysAccess.h) not including the string termination. Default 32. Note that changing this value may corrupt NVRAM. For NASYSACC_DEL, this must match the account password.

Argument	Description
level	Privileges assigned to the account. This can be any combination (binary OR) of the following: ■ NASYSACC_LEVEL_R — Minimum account privileges (login and read) ■ NASYSACC_LEVEL_RW — Full account privileges (login, read, and write) ■ NASYSACC_LEVEL_HTTP_R*n* — HTTP security realm 1–8 capability (for example, NASYSACC_LEVEL_HTTP_R1 to specify access for security Realm 1) ■ NASYSACC_LEVEL_GATEWAY — Access to the secure gateway ■ NASYSACC_LEVEL_SNMP_R — SNMP public community string definition ■ NASYSACC_LEVEL_SNMP_RW — SNMP private community string definition ■ NASYSACC_LEVEL_ROOT — All privileges
ipAddr	Pointer to an IP address, in dotted notiation (*a.b.c.d*), specifying that the account can be accessed only from that adress. If you do not want to filter the IP address, set to NULL.

Return values

Return value	Description
0	Success
NASYSACC_INVALID_OPERATION	*op* argument invalid — must be NASYSACC_ADD or NASYSACC_DEL
NASYSACC_INVALID_USERNAME	*user* pointer is NULL, or the string length exceeds NASYSACC_STRLEN_USERNAME
NASYSACC_INVALID_PASSWORD	*password* pointer is NULL, or the string length exceeds NASYSACC_STRLEN_PASSWORD, or the password does not match of a NASYSACC_DEL operation
NASYSACC_DUPLICATE_USERNAME	In a NASYSACC_ADD operation, *user* has already been used in another account
NASYSACC_INVALID_IPADDRESS	*ipAddr* is not pointing to a valid dotted notation address
NASYSACC_INVALID_ACCOUNT	In a NASYSACC_DEL operation, the account or capability does not exist
NASYSACC_DATABASE_FULL	In a NASYSACC_ADD operation, the number of accounts is already at the NASYSACC_MAX_ACCOUNTS
NASYSACC_NOMEM	Cannot allocate memory for the database

Index

A

B

C

D

L

M

N

O

P

R

S

NetSilicon

A Digi International® Company