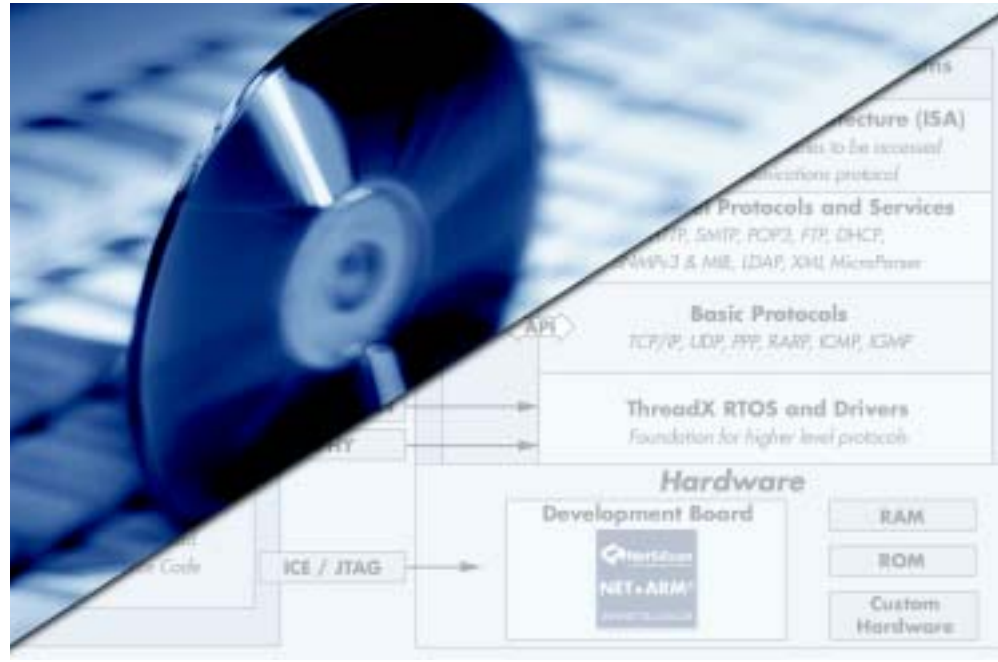
NetSilicon
A Digi International® Company

NET+OS User's Guide



NET+OS 5.0
8833237G

NET+OS User's Guide

Operating system/version: NET+OS 5.0
Part number/version: 8833237G
Release date: July 2002
www.netsilicon.com

Contents

Using This Guide

Review this section for basic information about this guide as well as for general support contact information

About this guide

This guide describes NET+OS and how to use it as part of your development cycle. Part of the NET+Works integrated product family, NET+OS is a network software suite optimized for the NET+ARM chip.

Who should read this guide

This guide is for software engineers and others who use NetWorks for NET+OS.

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

- Be familiar with installing and configuring software.
- Have sufficient user privileges to do these tasks.
- Be familiar with network software and development board systems.

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.
bold, sans serif type	Menu commands, dialog box components, and other items that appear on-screen.
Select **menu → option**	Menu commands. The first word is the menu name; the words that follow are menu selections.
`monospaced type`	Filenames, pathnames, and code examples.

What's in this guide

This table shows where you can find information this guide:

To read about	See
The components of NET+Works	Chapter 3, "NET+Works Introduction"
Application files, BSP files, and application entry points	Chapter 4, "NET+OS Applications"
Establishing application configuration settings	Chapter 5, "Configuring an Application"
Setting board support package "BSP" configuration parameters	Chapter 6, "Configuring the NET+OS Board Support Package"
Configuring the development board	Chapter 7, "Reserializing the NET+Works Board"
Using the components of the NET+ARM Windows-based Utility	Chapter 8, "Using the NETARM Windows-Based Utility"
Using the MIBMAN utility	Chapter 9, "Using MIBMAN"
Using the PBuilder utility	Chapter 10, "Using the Advanced Web Server PBuilder Utility"
Diagnosing problems	Chapter 11, "Troubleshooting"

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 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 Application Software Reference Guide* describes the NET+OS software application programming interfaces (APIs).
- *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.
- See the NET+Works hardware documentation for information about 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: 800 243-2333 / 781 647-1234 Fax: 781 893-1388 E-mail: 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.

NET+Works Introduction

CHAPTER 1

This chapter provides an overview of NET+Works.

Overview

The NET+Works products offer an embedded solution for hardware and networking software that is being implemented into product designs.

NET+Works is designed as an alternative to a PC network for products that must be connected to Ethernet for TCP access and for Internet/Intranet access. The NET+Works package includes:

- The NET+ARM chip
- A development board and board support package
- The networking firmware
- The object code with application programming interfaces (APIs)
- The NETOS program

For information about the NET+ARM chips and firmware, see the hardware documentation for the chip you are using.

Board support package

The board support package (BSP) is intended to work with the NET+OS real-time operating system. Some NET+Works drivers provide the necessary application programming interfaces (APIs).

NETARM Windows-Based utility

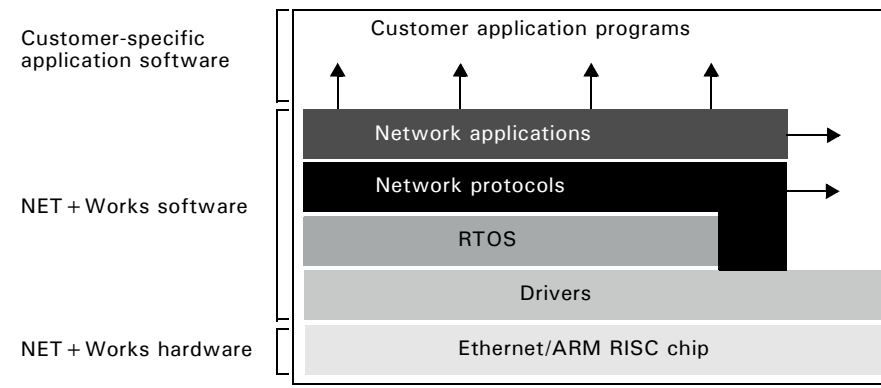
The NETARM Windows-Based utility, which is provided on the CD-ROM that came with your package, provides these utilities:

Utility	Purpose
HTML-to-C Converter	Converts HTML and related files to C code, which you can compile and link for an application

Utility	Purpose
Pad File utility	Extends a binary program image so it lines up evenly when you download the image to the development board
FTP Download utility	Lets you download a new flash ROM image to the development board
Application Wizard	Lets you customize a new application environment using a GUI
PBuilder Helper	Provides a command prompt with a search path for `pbuilder.exe` setup

You use the NETARM utility as you develop your application program. Most importantly, the toolkit provides a special compiler that translates your HTML code into C code. You can then compile the C code as part of the project. This allows you to create the pages necessary for your menu system to perform the tasks necessary for your device. Sample code is also provided.

This illustration shows how this part of the development cycle fits into the overall NET+Works functional overview:



NET+OS Applications

CHAPTER 2

This chapter describes the application files, BSP files, and application entry points provided with NET+OS.

Application files

Applications consist of the source code written by developers and the support files needed by the Green Hills™ tools. At a minimum, NET+OS applications consist of these files:

Filename	Description
appconf.h	Contains project-specific configuration parameters
debug.bld	Creates an executable that can be debugged in RAM on the target
debug.lx	Contains linker directives for generating the debug image
project.bld	Contains application-specific build commands
rom.bld	Creates a binary image that can be burned into ROM on the target
rom.lx	Contains linker directives for generating the ROM image
romzip.bld	Builds compressed application and code expansion; ENABLE_FLASH is defined here
romzip.lx	Linker command file to image compression/expansion
loader.c	Provides application configurations to expansion code; used in romzip.bld
ramimagezip.bld	Builds compressed applications
ramimagezip.lx	Linker command file for compressed application image
root.c	Contains the entry point for the application

Each file is described in the next sections.

appconf.h file

The appconf.h file defines constants that set configuration parameters for the application. The configuration settings are described in Chapter 3, "Configuring an Application."

bld and .lx files

Files with the `.bld` extension are build files used by the Green Hills MULTI® IDE. These files replace make files. Build files list the source files in a project and contain application-specific configuration settings that the MULTI development environment uses. Use the MULTI IDE to edit build files. For more information about build files, see the Green Hills documentation.

Files with the extension `.lx` contain directives for the Green Hills Linker. These directives control where the linker places the code and data segments.

To change linker command files, use a text editor.

debug.bld and debug.lx files

The `debug.bld` and `debug.lx` files create an executable version of the application that you can download into RAM on the target board and debug with the MULTI debugger.

▶ **To prepare an application for debugging:**

1. Open the application's `debug.bld` file in the MULTI IDE.
2. Use MULTI's build function to create a debuggable version of the file.

You can download the executable image to the target board and debug it with the MULTI debugger.

rom.bld and rom.lx files

The `rom.bld` and `rom.lx` files create a binary image of the application, which can be written into ROM on the target.

To create a ROM image, load the application's `rom.bld` file in the MULTI IDE.

You can use the flash ROM update utility to write the image header into flash on a NET+ARM development board.

romzip.bld and romzip.lx files

The `romzip.bld` and `romzip.lx` files create a compressed binary image of the application with the expansion, which can be written into ROM on the target.

▶ **To create a compressed ROM image with expansion code:**

1. Load the application's `romzip.bld` file in the MULTI IDE.
2. Using MULTI's build function, generate the compressed ROM image.

You can use the flash ROM update utility to write the image `romzip.bin` into flash on a NET+ARM development board.

loader.c file

The `loader.c` file provides application configurations to expansion code used in `romzip.bld` to minimize the code size for compressed code expansion.

ramimagezip.bld and ramimagezip.lx

Use the `ramimagezip.bld` and `ramimagezip.lx` files to build the compressed image, which is included in `romzip.bld`.

The `project.bld` file is the main build file for the application. The information in this file is independent of whether the application is built for debugging or built as a RAM image. The build file is a subproject of the `debug.bld`, `rom.bld` and `ramimagezip.bld` files.

Place changes to the application that affect both the ROM image and debug executable in the `project.bld` build file. To add new source and header files to the project, list them in `project.bld`.

root.c file

The `root.c` file contains the `applicationStart` and `applicationTcpDown` functions. These functions, which are the entry points into the application, are called by the kernel during initialization.

BSP files

In addition to the application files, these files in the BSP directory are compiled:

This file	Contains
bsproot.c	Kernel startup code
dialog.c	Configuration dialog boxes
reset.s	Part of the vector table
decompress.c	Routines to decompress the zipped code in flash into RAM and restart from RAM

Because these files contain code that is affected by settings in appconf.h, they are compiled separately for each application.

Application entry points

All NET+OS applications must implement the applicationStart and applicationTcpDown functions, described in the next sections.

void applicationTcpDown(void)

The applicationTcpDown function is called once every timer tick by the kernel while it waits for the TCP/IP stack to start up. The TCP/IP stack may take a long time to start up, or it may not be able to start up at all. This can happen if you are using DHCP to get an IP address and one of these conditions exists:

- No DHCP servers are on the network.
- No IP addresses are available.

If the application does not want to handle the error, this function can just return. If the application handles the error, applicationTcpDown keeps track of how long it has been waiting for the IP stack to come up and handle the error if it times out. The function is called from a thread, and all kernel services — except those that use the TCP/IP stack — are available. This function must return.

This example demonstrates an implementation of `applicationTcpDown`:

```
void applicationTcpDown(void)

{
static int stackTimeout = 30 * BSP_TICKS_PER_SECOND;
static int ledTimer = 0, redLedOn = 0;

if (stackTimeout)          /* if we still have not timed out*/
{
          stackTimeout--; /* then keep decrementing it*/
}
else       /* if we have timed out*/
{          /* then blink the LED*/
if (ledTimer)             /* is it time to blink the LED?*/
          {
          ledTimer--;     /* if not, then just decrement timer*/
          }
          else            /* time to blink it*/
          {
          if (redLedOn)   /* if LED was on */
          {
          NALedRedOff();  /* turn it off*/
          }
          else            /* if LED was off*/
          {
          NALedRedOn();   /* turn it on */
          }
          redLedOn = 1 - redLedOn;/* invert LED bit and reset timer*/
          ledTimer = BSP_TICKS_PER_SECOND;
          }
   }
}
```

void applicationStart(void)

The `applicationStart` function creates and initializes any resources needed by the application and then starts the application.

This function is called once by the kernel from a thread after the kernel and TCP/IP stack have started. At this point, all kernel services, including the TCP/IP stack, are available. This function does not need to return.

Creating a new application

A template application is provided with the NET+OS BSP. This application implements a simple Hello World application. If NET+OS is installed in the default directory, the template application is located in:

Pathname	For
`c:\netos5_gh35\src\apps\template`	NET+OS with Green Hills 3.5
`c:\netos5_gh\src\apps\template`	NET+OS with Green Hills 2.1
`c:\netos5_gnu\src\apps\template`	NET+OS with GNU Tools

Using the template application

▶ To create a new application:

1. Copy the template application into a new subdirectory of `c:\netos4\src\apps`.
2. Modify the code in `root.c` as appropriate for the new application.
3. Add new source files by editing the `project.bld` file.

You can edit the application by editing the `appconf.h` file.

Root task

The kernel creates a task to call the `applicationStart` and `applicationTcpDown` application functions. This is a standard thread and can be used by the application as one of its own threads.

Configuring an Application

CHAPTER 3

This chapter describes how to establish configuration settings.

Overview

This chapter discusses how to establish these application configuration settings:

- User-defined error handling
- Root task parameters
- Networking parameters
- TCP/IP stack buffers
- Console I/O port
- Power-on self tests (POST)
- Security

Setting a user-defined error handler

The NET+OS Board Support Package (BSP) calls an error handler when it encounters a fatal error. This error handler normally blinks the LEDs on the development board in a pattern that indicates the type of error that occurred. You can replace the default error handler with a user-defined error handler.

Define the replacement error handler in this way:

```
void userErrorHandler (char *description, int redBlinks, int
   greenBlinks)
```

The `description` parameter points to a text description of the error condition. The `redBlinks` and `greenBlinks` parameters indicate the blink pattern that would be used in the default error handler. The default error handler is executed if the user-defined error handler returns.

▶ **To install a replacement error handler:**

1. Set the constant `APP_ERROR_HANDLER` in `appconf.h` to point to the user-defined error handler.
2. Rebuild the application.

Setting root task parameters

You can set the stack size and the priority level of the root task:

To set	Set the constant
The stack size of the root task	APP_ROOT_STACK_SIZE in appconf.h
The priority level of the root task	APP_ROOT_PRIORITY in appconf.h

Setting networking parameters

To communicate on an IP network, each system needs an IP address, subnet mask, gateway, and MAC address. On the development board, the MAC address is derived from the board's serial number.

To have NET+OS prompt for these parameters with an interactive dialog box, set the APP_DIALOG_PORT constant in appconf.h to the name of a defined serial port. During startup, NET+OS prompts the user to enter these values on the selected serial port. The user can either use DHCP to get an IP address, subnet mask, and gateway address or enter these values manually. The user also is prompted to set the serial number (and therefore, the MAC address) of the development board.

Depending on the network settings, one of three results occurs:

- If the APP_USE_NVRAM constant in appconf.h is set, these values are stored in NVRAM on the board.
- If APP_DIALOG_PORT is not set but APP_USE_NVRAM is set, NET+OS uses the values stored in NVRAM without prompting the user with a dialog box.
- If neither APP_DIALOG_PORT nor APP_USE_NVRAM is set, NET+OS uses the values set in the APP_IP_ADDRESS, APP_IP_SUBNET_MASK, APP_IP_GATEWAY, and APP_BSP_SERIAL_NUM constants in appconf.h for the IP address, subnet mask, gateway, and serial number respectively.

This table is a summary of the network settings and the result returned for each:

APP_DIALOG_PORT	APP_USE_NVRAM	Result
Set	Set	Prompt with a dialog box, and save values in NVRAM
Set	Not set	Prompt with a dialog box
Not set	Set	No dialog box; use the values stored in NVRAM
Not set	Not set	Use the values set in constants defined in `appconf.h`

`APP_FILE_SYSTEM_SIZE` in `appconf.h` determines the maximum number of simultaneous open files in the file system of the Advanced Web server (AWS). The default value is 9, which is the number of connections AWS allocated.

Be aware that increasing `APP_FILE_SYSTEM_SIZE` reduces the available memory.

Configuring TCP/IP stack size

TCP/IP stack allocates packet buffers out of the network *heap* — memory used for all TCP/IP stack dynamic memory allocations. To change the size of the network heap, change the `APP_NET_HEAP_SIZE` constant in `appconf.h`. In this case, you also might need to change the size of the application heap defined in the `debug.lx`, `rom.lx`, or `ramimagezip.lx` file.

There are no hard and fast rules for how many TCP/IP buffers to allocate. You'll need to experiment with values and find a set that works well with your application. In general, the more buffers, the better.

Setting the console I/O port

You can direct the `stdin`, `stdout`, and `stderr` file handles to a NET+OS serial port. This has the effect of sending the output of `printf` statements out the serial port, which can be useful for debugging.

To direct the file handles to a serial port, set the constant APP_STDIO_PORT in appconf.h to the name of a serial port. If APP_STDIO_PORT is not set, the output of printf statements is either discarded or put into the debugger's I/O window if the ICE in use supports I/O redirection.

Enabling the POST

NET+OS includes a set of Power-On Self Tests (POSTs) that test the functionality of the NET+ARM chip and the development board. If you set the constant APP_POST to 1 in the root.c file, these POSTs are performed at startup.

For more information on POSTs, see the *NET+OS BSP Porting Guide*.

Configuring security parameters

This section describes the security parameters you can set to:

- Define the root account and password
- Define the administrators account and password
- Define a fixed static encryption key
- Enabling SNTP configuration
- Enabling Kerberos configuration

Defining the root account and password

The NET+OS startup adds a root account if the compiler directive APP_ROOT_PASSWORD is defined in appconf.h. When the root account username *root*, which has all privileges, is created at startup, it has a default password of the string defined by the APP_ROOT_PASSWORD compiler directive. This password is limited to a length of less than NASYSACC_STRLEN_PASSWORD, defined in the sysAccess.h file, and is set to 32 bytes.

When this account is defined, the interactive dialog console (active when APP_DIALOG_PORT is defined) is root password protected. Updating the dialog parameters requires the root password.

After the root password is provided, the dialog console provides an entry to update and change the root password.

Because the root password can be extracted from the application binary image, Netsilicon strongly recommends that you update the root password on all systems soon after the system is set up.

Never use the root password over insecure applications such as Telnet or FTP.

Defining the administrators account and password

The NET+OS startup adds an administrators account if the compiler directive APP_ADMINISTRATORS_ACCOUNT is defined in appconf.h. When defined, the administrators account, which has NASYSACC_LEVEL_GATEWAY privileges, has a username and password defined as the APP_ADMINISTRATORS_ACCOUNT compiler directive. This username and password is limited to a length of less than NASYSACC_STRLEN_PASSWORD, defined in the sysAccess.h file, and set to 32 bytes.

When this account is defined, the interactive dialog console (active when APP_DIALOG_PORT is defined) provides an entry to update and change the administrator's account password. Because the account can be extracted from the application binary image, Netsilicon strongly recommends that you update the administrators account password on all systems soon after the system is set up.

The administrator's account can be used only for Username/Password key exchange on the NET+OS Secure Proxy. This is useful only with NET+OS Security Module.

Defining a fixed static encryption key

The NET+OS configuration manager maintains a static fixed encryption key if the compiler directive APP_FIXED_GATEWAY_KEY is defined in appconf.h. When APP_FIXED_GATEWAY_KEY is defined, this key can be extracted through the call NASecGetSystemFixedKey (see NASecurity.h), and used as an encryption key with a peer who also shares knowledge of this key.

The default value is defined as the APP_FIXED_GATEWAY_KEY compiler directive, and is limited to a length of less than NASYSACC_STRLEN_PASSWORD, defined in the sysAccess.h file, and presently set to 32 bytes. This functionality is useful only with NET+OS Security Module.

When the compiler directive APP_FIXED_GATEWAY_KEY is defined in appconf.h, the interactive dialog console (active when APP_DIALOG_PORT is defined) provides an entry to update and change the key. Because the key can be extracted from the application binary image, Netsilicon strongly recommends that you update fixed keys on all systems soon after the system is set up.

Enabling SNTP configuration

The NET+OS configuration manager maintains a list of two candidates Simple Network Time Protocol (SNTP) Servers, when the compilation directive APP_USE_NETWORK_TIME_PROTOCOL is defined.

The servers are stored in either NVRAM (when APP_USE_NVRAM is defined) or taken directly from the appconf.h header from compilation directives APP_SNTP_PRIMARY and APP_SNTP_SECONDARY.

The servers can be retrieved using either the NAGetSntpPrimaryServer or NAGetSntpSecondaryServer calls. These functions are useful when you call NAstartSntp, which is in NASntp.h. When the APP_USE_NETWORK_TIME_PROTOCOL directive is defined, the interactive dialog console (active when APP_DIALOG_PORT is defined) provides an entry to update and change the two servers.

This functionality is available only with the NET+OS Security Module or the NET+OS SNTP Module.

Enabling Kerberos configuration

The NET+OS configuration manager maintains parameters to be used for a Kerberos client, if the compiler directive APP_USE_KERBEROS is defined in appconf.h. The parameters include the Kerberos KDC IP Address, realm name, username and password.

These parameters are either stored in NVRAM (when APP_USE_NVRAM is defined) or taken directly from the appconf.h header. When the appconf.h definitions are used, values must be defined for directives APP_KERBEROS_REALMNAME, APP_KERBEROS_KDC, APP_KERBEROS_USERNAME, and APP_KERBEROS_PASSWORD. These parameters can be retrieved when APP_USE_KERBEROS is defined, using the Kerberos function NAKrb5GetConfigParameters. This functionality is useful only with NET+OS Security Module.

Configuring the NET+OS Board Support Package

CHAPTER 4

This chapter describes the tasks for configuring BSP parameters.

Introduction

This chapter describes how to set BSP configuration parameters. The constants described in this chapter are defined in the `bspconf.h` file. If NET+OS was installed in the `c:\netos5\`*`installation directory`*, this file is located in the directory `c:\netos5\`*`installation directory`*`\h`.

For more information about the board support package, see the *NET+OS BSP Porting Guide.*

Configuring the BSP

To configure the BSP, you need to set these parameters:

- The system clock rate
- Device drivers
- Ethernet driver parameters
- TCP/IP parameters
- The starting location of ROM, RAM, and NVRAM
- The size of NVRAM
- The crystal speed
- The number of flash memory banks to use

Setting the system clock rate

NET+OS has a system clock that is used for timing and for controlling thread preemption.

To set the clock rate, set the constant `BSP_TICKS_PER_SECOND`.

Selecting device drivers

You can choose which standard NET+OS device drivers are available by defining or undefining these constants:

- BSP_INCLUDE_SERIAL_DRIVER
- BSP_INCLUDE_PARALLEL_DRIVER
- BSP_INCLUDE_NUL_DRIVER
- BSP_INCLUDE_LOOPBACK_DRIVER

For example, if you undefine the constant BSP_INCLUDE_LOOPBACK_DRIVER, the BSP is built without the loop back driver.

Setting the number of Ethernet receive buffers

The Ethernet driver uses a thread to move packets from the driver's receive queue to the TCP/IP stack.

The number of receive packet buffers used by the Ethernet driver is defined in the efe_mod.h file as #define MAX_RX_BUFFERS 64. These buffers hold received packets until the Ethernet receiver thread passes them up to the TCP/IP stack. You can increase the number of receive packet buffers to a maximum of 128.

Receive buffers are allocated from the network heap. The size of the heap memory needed for each buffer is 1756 bytes.

Setting the starting location of ROM, RAM, and NVRAM

The standard NET+OS BSP controls the location of ROM, RAM, and NVRAM by programming chip selects on the NET+ARM chip.

Set the location for these devices by defining these constants:

- BSP_ROM_BASE
- BSP_RAM_BASE
- BSP_NVRAM_BASE

Setting the size of NVRAM

To set the size of the NVRAM device, define the BSP_NVRAM_SIZE constant.

Setting the crystal speed

The frequency of the crystal connected to the XTAL signal on the NET+ARM chip is indicated by the compiler directive `CRYSTAL_OSCILLATOR_FREQUENCY`, which is defined in `/h/sysclock.h`. Normally, you use an 18.432 MHz crystal. Using a different crystal significantly affects the BSP. For more information, see the *NET+OS BSP Porting Guide*.

Setting the number of flash memory banks

You can set the number of flash memory banks to use by changing the `BSP_FLASH_BANKS` constant. This constant informs the flash driver of the number of flash memory banks that are available. The default value is 1. Multiple flash memory banks must be continuously addressable.

Reserializing the NET+Works Board

CHAPTER 5

This chapter describes how to configure the NET+Works development board.

Overview

The NET+Works development board ships with a boot ROM application programmed in flash ROM. The boot ROM application allows you to configure the board.

This chapter provides instructions for both the Raven and the JEENI.

Development boards and serial numbers

Each development board has a unique, factory-set serial number stored in NVRAM. The serial number also is printed on a sticker on the board. Because the Ethernet MAC address is determined from the serial number, it is important that you assign each board a unique serial number. The serial number must have eight digits. For serial numbers with fewer than eight digits, add leading zeros. For example, if your number is 123456, then assign 00123456 to the board. If the contents of NVRAM are lost, 99999999 is used as a default serial number. In this case, you need to reserialize the board.

Observing the LEDs

Be aware of the amber and green LEDs whenever you power cycle the development board. The LEDs provide information about the board that allows you to monitor the status of the board at all times.

Preparing to reserialize

Before you reserialize the development board, start a HyperTerminal session. Keep the HyperTerminal window open during all your testing. If your system resources are limited, keep the HyperTerminal window open only when you power cycle the board.

When you start a HyperTerminal session, you see a display similar to this one:

```
NET+WORKS Version 5.00
Copyright (c) 2002, NetSilicon, Inc.
-----------------------------------------------------
NETWORK INTERFACE PARAMETERS:
   IP address on LAN is 132.74.179.148
   LAN interface's subnet mask is 255.255.255.0
   IP address of default gateway to other networks is
132.74.179.587
HARDWARE PARAMETERS:
   Serial channels will use a baud rate of 9600
   This board's serial number is 99994336
   This board's MAC Address is 0:40:af:79:61:0
   After board is reset, start-up code will wait 5 seconds
-----------------------------------------------------
Press any key in 5 seconds to change these settings.
```

Reserializing a development board

▶ To reserialize a board:

1. Connect a terminal or terminal emulator to serial port 1 on the development board.
2. Power up the board.

   You see the board's current configuration on the terminal emulator.
3. Press **Enter** before the timeout expires.
4. To modify the configuration settings, press any key.
5. Either set the networking parameters or enable DHCP, then set the serial baud rate and the time-out for the dialog box when you are prompted.
6. When you are prompted to set the serial number, click **Yes**.
7. Type the 8-digit serial number printed on the board. (See the label on the board).
8. Type **C** and press **Enter** to continue. The new configuration settings are written into NVRAM.

Restoring the contents of flash ROM

NET+Works development boards ship with a boot ROM program written in flash ROM. The boot ROM program implements support for debugging and provides an FTP server that can be used to update flash ROM.

If the flash ROM gets corrupted, you cannot use either a debugger or the built-in FTP server to restore the boot ROM. Also, the ARM processor may be put into an invalid state when it executes the contents of flash ROM. In this case, the in-circuit emulator (ICE) you are using may not be able to communicate with the processor.

You can restore the original boot ROM program by following a procedure in which you:

- Configure the target development board and ICE
- Build the boot ROM image from the NAFTPROM application
- Build the FTP Flash download program from the NAFTPAPP application
- Run the NET+ARM Windows-based utility
- Verify boot ROM Image on target development board

These steps are described in the next sections. Be aware that the order of these tasks is important; you *must* do the tasks in the order in which they are presented.

Step 1: Configuring the development board and the ICE

1. Connect the ICE you are using — either the Raven or the JEENI — as described in the *NET+OS with Green Hills Getting Started Guide* or the *NET+OS with GNU Tools Getting Started Guide*.
2. Connect target serial port 1 to the communications port of your computer.
3. Power up the target development board.
4. Power up the ICE.
5. Set the target development board DIP switch settings like this:
   - Big Endian mode (`LEND = OFF`)
   - Disable Flash (`FL=OFF`)
6. Start a HyperTerminal session.

Step 2: Building the boot ROM image from naftprom

1. Start Green Hills MULTI IDE
2. Select **File → Open Project in Builder.**
3. Type `c:\netos5_`*`installation directory`*`\src\examples\naftprom\32b\rom.bld`.
4. Select **Build → Rebuild All.**
5. Select **File → Exit All.**

Step 3: Building the FTP flash download program from naftpapp

In this part of the procedure, you send the FTP utility to the development board:

1. Start the Green Hills MULTI IDE.
2. Select **File → Open** `c:\netos5_`*`installation directory`*`\src\examples\naftpapp\32b\naftpapp.bld`
3. Select **Build → Build All.**
4. To start a debug session, select **DEBUG → DEBUG naftpapp**.
5. Set up a debug session in the Command window.

   Use the playback command (<) to execute `break.ini`:

   `<c:\netos5_`*`installation directory`*`\break.ini`
6. To start the download, click **Go**.

   You see a message that indicates the download is underway.

   When the download process is completed, you see messages such as these:

   ```
   Download complete."
   Running c:\netos5_installation directory \src\ examples
   \naftpapp\32b\naftpapp"
   Stopped by breakpoint"
   ```

   A breakpoint occurs at the source code line

   `ftp_init_flash_download;`
7. To continue working, click **Go**.

   Do not terminate the connection until you have transferred files and the session is over.

At this point, a network pathway between the board and your computer is established.

Step 4: Running the NETARM Windows-based utility

In this step, you use the NETARM Windows-based utility to send `rom.bin` to the development board.

1. To start FTP Download in the NETARM Windows-based utility, select **Start Programs → NET+OS5_*installation directory*→ NET+OS.**
2. Enable flash (FL=ON).
3. Click the FTP icon.

   You see the FTP Download window.
4. Select local directory `c:\netos5_`*`installation directory`*`\src\examples\naftprom\32b`.
5. To check that the previous procedure worked, select the `rom.bin` file.
6. Set the IP address of the remote unit. This address must match the IP address of the target board.
7. Click **Download**.
8. When the download is finished, click **Close**.

Step 5: Verifying the boot ROM Image on the development board

1. Power off the development board and ICE.
2. If you exited from HyperTerminal, start a new session.
3. Power up the target development board.

   You see the updated information about the board.
4. To modify the information, press any key within ten seconds.
5. Type **M** and press Enter.

   You see prompts to modify the information.
6. Make the changes as you are prompted.

Using the NETARM Windows-Based Utility

CHAPTER 6

This chapter describes the NETARM Windows-Based utility, a set of utilities and applications you'll use as part of your development effort.

Overview

The NETARM Windows-Based utility is made up of several utilities you use as part of your development effort. Depending on the in-circuit emulator (ICE) you are using, you can use some or all of of these utilities. This table shows the utilities, a description of each, and which ICE you can use with each utility:

Utility	Description	ICE
HTML-to-C Converter	Converts HTML files, images, Java applets, and audio files into C code	Green Hills GNU
Pad File utility	Lets you line up a flash ROM image evenly when you download the image to the development board	Green Hills only
FTP Download utility	Lets you download new or updated software to the development board for testing	Green Hills GNU
Application wizard	Lets you customize a new application environment using a GUI (	Green Hills only
PBuilder Helper	Provides a command prompt with a search path	Green Hills GNU

You start the utilities from the main window of the NETARM Windows-based utility:

HTML-to-C Converter (Green Hills and GNU)

The HTML-to-C Converter converts HTML and related Web page files to C code so the Web pages can be compiled and linked for an application.

The HTML-to-C converter provides an easy way to integrate Web pages into the Web server. This utility translates your Web pages into standard C source files so you can compile them with a compiler. In addition to HTML pages, you can incorporate images, Java applets, and sound files into your Web server. Components converted to C code are easily integrated into NET+Works software applications. Web components in this context are defined as:

- HTML pages
- GIF or JPEG images
- Java applets
- Audio files

This section provides background information on HTML and describes how to use the HTML-to-C Converter and several of the utility menus.

Background

Web content consists of HyperText Markup Language (HTML) pages, images (such as .gif, .jpeg, and so on), and applets that are sent by a device in response to HyperText Transport Protocol (HTTP) requests from Web browsers.

To incorporate Web content into an embedded device, an HTTP server is first built into the device using the APIs provided with NET+Works (see the *NET+OS Application Software Reference Guide*). The server processes HTTP requests and responds with Web content.

The next step incorporates the Web content into the HTTP server. Commercial Web servers installed on UNIX or Windows NT operating systems have storage disks with large file systems. This file system makes incorporating Web content fairly routine because pages and images are added to an existing directory, making the files Web-accessible. Embedded devices normally have read only memory (ROM) without a file system. The Web content in such cases must be incorporated directly into the embedded device application stored in ROM.

When you write HTML from scratch, you can develop HTML pages by adding HTML markup tags to the text content, using either a text editor or Web authoring tool. The same pages can be added to an embedded device by writing application code to physically return the HTML page. The page is stored in a large character

buffer in the device and is returned through a network application programming interface, such as sockets.

The sockets method for storing Web pages has limits. As tools for generating HTML pages become more advanced, webmasters do not generate HTML pages by hand because a Web authoring tool is much more efficient and removes a significant amount of typing in markup tags. *Static* HTML pages, those pages with content that never changes, are only a small part of the Web content provided by a Web server. *Dynamic* content, where the Web content changes over time, is commonplace and necessary for status reporting. *Forms processing*, which accepts user input and acts on it, is also commonplace and necessary to make configuration changes in an embedded device. An HTML generation or Web authoring tool does not solve the problem of providing dynamic content or forms processing. Application code in the commercial Web server (written in PERL, C++, or Java) is necessary for these types of Web content.

Incorporating static and dynamic content and forms processing

The problem for embedded devices is how to incorporate static content, dynamic content, and forms processing into the embedded HTTP server source code. The solution to this problem parallels that for commercial Web servers.

The HTML-to-C Converter automatically converts Web content into application source code:

- Static pages are converted into the necessary program calls to send back HTML, image, and applet content that does not change over time.
- Dynamic content has proprietary non-HTML markup tags inserted into the HTML source using an HTML editor. The HTML-to-C utility recognizes these tags and produces shell routines and calls to the routines in the application source code. The embedded designer is then responsible for implementing the routine, so the appropriate dynamic content is returned when the routine is called.

This utility also recognizes HTML form tags and adds the necessary shell routines to be called when forms data is sent back to the embedded Web server.

The embedded Web server typically has an application programming interface (API) that makes it easy to retrieve the common gateway interface (CGI) data supplied by the browser in response to a forms submission. The embedded

designer is responsible for filling in the shell routine with the code necessary to handle the data and send back the reply.

Preparing to use the HTML-to-C Converter

The filename of an HTML URL is its full URL name. Because slash (/) is not a valid character for a file name, slashes are converted to an underscore (_). For example, `/abc/def.html` is generated as `_abc_def.c`.

Before you use this utility, create Web pages using an HTML editor. Then come back to this chapter.

▶ **To create an HTTP server:**

1. Create Web pages using an HTML editor.
2. Use the HTML-to-C Converter to convert your Web pages into C code.
3. If your Web pages contain dynamic content or forms, you need to add your own code to the generated C empty routines. (See "Types of HTML content" for a description of dynamic content and form processing.)
4. Put all your C source files into your application's `project.bld` file or Makefile.

How the HTML-to-C converter works

The HTML-to-C converter recognizes two file types:

- **Text files.** Any file with an extension of .html, .htm, or .txt. A text file causes the converter to generate a .c file with a similar file name.

  See "File name conventions" later in this section.

- **Binary files.** All binary files are converted and stored in the `bindata.c` file. All the URL information, including security, is stored in a file `url.c`.

  The filenames `bindata.c` and `url.c` are default names that you can change.

For example, say your Web server contains four files — `home.htm`, `x.htm`, `y.jpg`, and `z.gif` — and you want to organize your Web hierarchy (URLs) like this:

```
/home.htm
/x.htm
/images/y.jpg
/images/z.gif
/~John/home.htm
/~John/y.jpg
```

These C files are generated:

```
_home.c  (from url /home.htm)
_x.c  (from url /x.htm)
_~John_home.c (from url /~John/home.htm)
bindata.c (contains data for y.jpg and z.gif)
url.c
These entries are added to the URL Table in the file url.c:
URLTableEntry URLTable[] = {
“/home.htm”, 1, Send_function_0, (HSTypeFnHtml)HSTypeHtml, 0,
    Unprotected,
“/~John/home.htm”, 1, Send_function_1, (HSTypeFnHtml)HSTypeHtml, 0,
    Unprotected,
“/~John/y.jpg”, 0, Send_function_2, (HSTypeFnBinary)HSTypeJpeg,
    453, Unprotected,
“/images/y.jpg”, 0, Send_function_2,
    (HSTypeFnBinary)HSTypeJpeg, 453, Unprotected,
“/images/z.gif”, 0, Send_function_3, (HSTypeFnBinary)HSTypeGif,
    499, Unprotected,
“/x.htm”, 1, Send_function_5, (HSTypeFnHtml)HSTypeHtml, 0,
    Unprotected,
NULL  /* last entry MUST be NULL */
};
```

There is only one `home.htm` file, but because two URLs (`home.htm` and `/~John/home.htm`) refer to it, two files are generated. This lets you put different code to serve the same page (`home.htm`) under different absolute URLs.

Binary data, however, is treated differently because it doesn't need to be customized. That's why only one copy of the data is stored in `bindata.c` for `y.jpg`, even though it is referred to in both `/~John/y.jpg` and `/images/y.jpg`.

The URL table is being used in the AppSearchURL function. When a user clicks a URL link from a Web browser, the Web server does a search on the URL table. If the server finds the entry for the URL, it calls the corresponding function (for example, `Send_function_1`) to send back the page.

Here is an example of the function Send_function_1 in the file `_~John_home.c`:

```
void Send_function_1(unsigned long handle)
{
   HSSend (handle, “<html>\n”);
   HSSend (handle, “\n”);
   HSSend (handle, “<head>\n”);
   HSSend (handle, “<meta http-equiv=\”Content-Type\“\n”);
   HSSend (handle, “content=\”text/html; charset=iso-8859-
      1\“>\n”);
   HSSend (handle, “<meta name=\”GENERATOR\
      “content=\”Microsoft FrontPage Express 2.0\“>\n”);
   HSSend (handle, “<title>John’s Home Page</title>\n”);
   HSSend (handle, “</head>\n”);
   HSSend (handle, “\n”);
   HSSend (handle, “<body bgcolor=\”#FFFFFF\“>\n”);
   HSSend (handle, “\n”);
   HSSend (handle, “<h3>Welcome to John’s Home </h3>\n”);
   HSSend (handle, “\n”);
      HSSend (handle, “</body>\n”);
      HSSend (handle, “</html>\n”);
      }
```

Types of HTML content

The HTML-to-C Converter recognizes three types of HTML content:

- Static
- Dynamic content
- Input form

Static

On a *static page,* the content of the page is constant. Nothing needs to be added after the page is converted.

Dynamic content

Sometimes you want a Web page to look different every time you access it. For example, you may want to write a page that reports the temperature of an oven. This is where dynamic content is useful. All you have to do is to embed a special tag — `_NZZA_` — into your HTML file. The HTML-to-C converter understands this tag and generates an empty routine at the end of the generated C file.

This example is a portion of the Web page that reports the temperature:

```
<h1> The temperature of the oven is _NZZA_get_temp </h1>
```

This is translated into statements:

```
HSSend (handle, "<h1> The temperature of the oven is");
na_get_temp(handle);
HSSend (handle, "</h1>");
```

Note that `_NZZA_get_temp` is being converted into the function call `na_get_temp`, which is located at the end of the file:

```
void na_get_temp(unsigned long handle)
   {
   }
```

You need to add real code to get and format the temperature, and then send it back to the browser. For example:

```
void na_get_temp(unsigned long handle)
   {
      char buf[100];
```

```
    int temperature;

    temperature = get_oven_temp();
    sprintf (buf, "%4d", temperature);
    HSSend (handle, buf);
}
```

Input form

One use of an embedded Web server is collecting data (for example, configuration information) from a user. The easiest way to do this is with an input form. The HTTP Server library API `HSGetValue` is used to capture data collected in this form.

To use forms, you need to include an action field in your form; for example:

```
<FORM ACTION="ip_config" method ="POST">
```

This field allows the HTML-to-C converter to generate an empty function that must be filled in to collect the data sent from the browser. `Post_` is prefixed to the name of the action.

The generated function name must be a valid C identifier, so your action name must not contain any illegal characters. For example, `Post_ip_config` is added at the end of the file:

```
void  Post_ip_config (unsigned long handle)
{
}
```

To extract data sent with the form, use `HSGetValue`:

```
void  Post_ip_config (unsigned long handle)
{
          char ipaddr[32];

   /* assume the data is a text input, with name
     attribute set to IP_ADDR */
   HSGetValue (handle, "IP_ADDR", ipadddr, 32);
   …
   /* at this point, whatever the user
      input will be stored in ipaddr   */
}
```

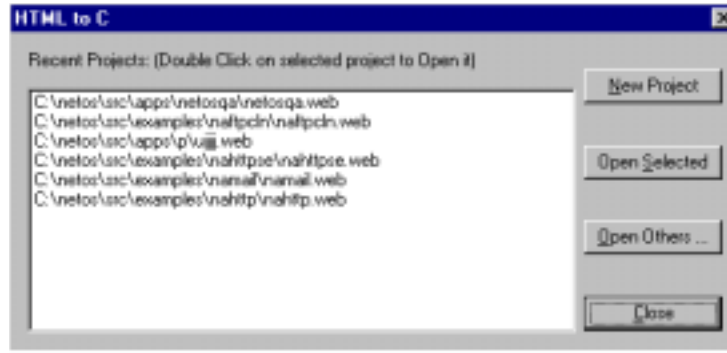
Using the HTML-to-C converter

Files for the same Web server application are organized as projects. A project file has a file extension of `.web`.

▶ To use the HTML-to-C utility:

1 Select **NETARM Windows-based Utility → HTML-to-C**.

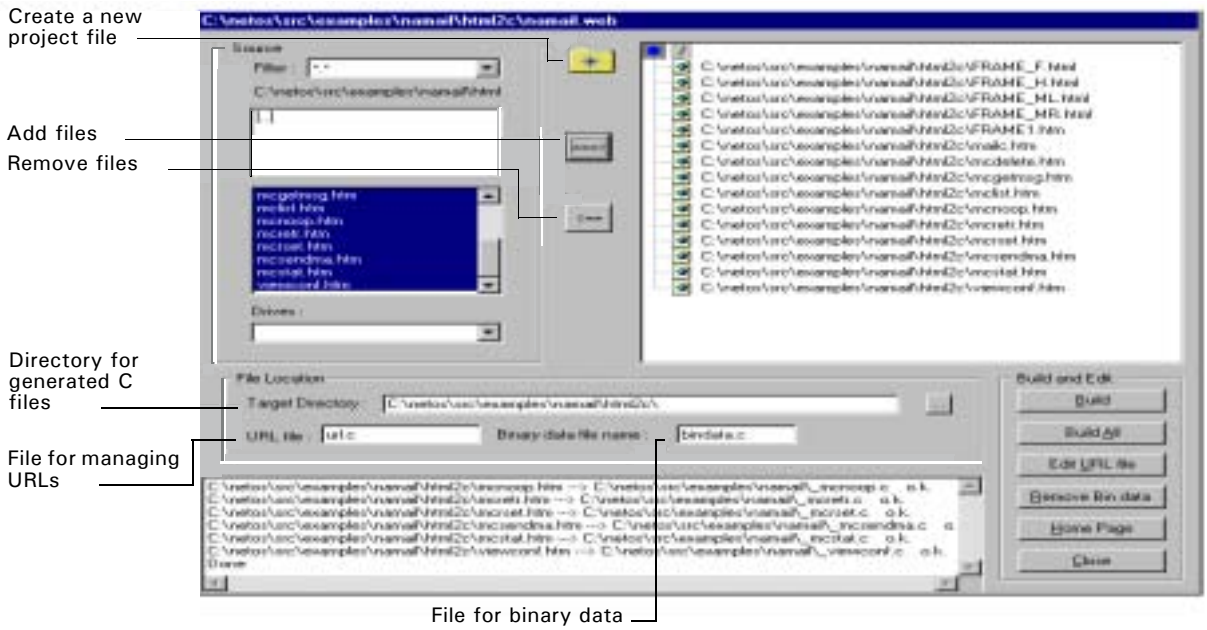
You see this dialog box:



2 Either open an existing project or create a new one:

To	Do this
Open an existing project	Do one: ■ If you see the project on the list, select the project and click **Open Selected**. ■ If you don't see the project on the list, click **Open Others**, navigate to and select the project, and click **Open Selected**.
Create a new project file	Click **New Project,** type a name for the project, and select a location.

You see this dialog box:



From this dialog box, you can:

- Generate C source files (only those that have changed)
- Regenerate C source files for all files
- Add or remove source files
- Edit the `URL.c` file
- Specify the locations for generated C files, binary data, and the URL table
- Set or change the project's home page

File location fields

- **Target Directory.** Type the name of the directory in which to put the generated C files.
- **URL file.** Type the name of the file that contains the URL table. The default name is `url.c`.

- **Binary data file name.** Type the name of the file that contains all binary data. The default name is `bindata.c`.

  If your project does not have any binary files, you don't need to include this file in your `.bld` file or Makefile.

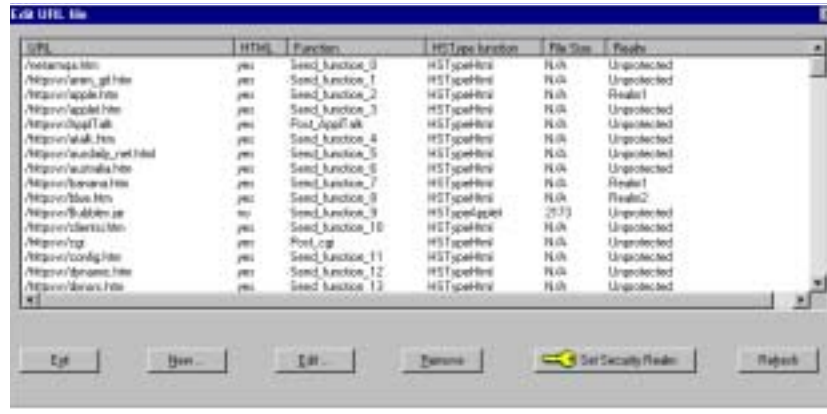
Build and edit section

This table describes the Build and Build All buttons:

Button	Description
Build (Generate C) files	Generates C source files. Regenerates only the files (.html, and so on) that have changed since the last build
Build All	Regenerates all files

Editing URL files

Use Edit URL file to edit the URL table in the `url.c` file. You can add a new URL, change attributes (such as adding a realm to an existing URL), or remove an obsolete URL.



The functions at the bottom of the Edit URL file dialog box let you generate and maintain files. The Set Security Realm function is described later in this section.

Removing obsolete data

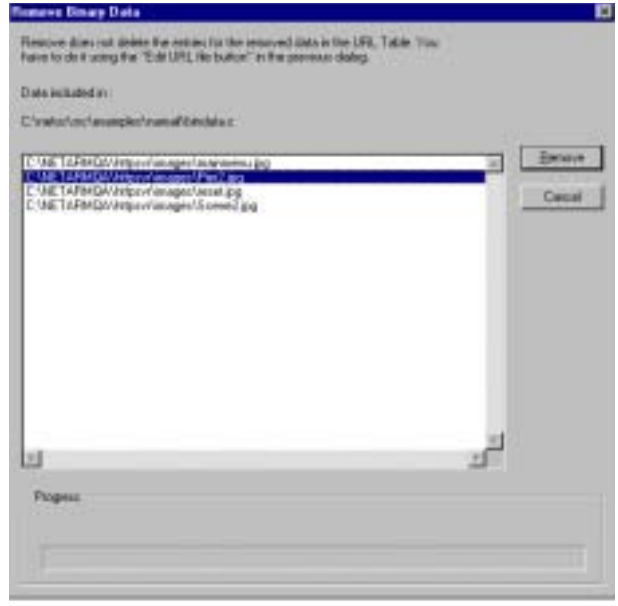
Removing obsolete data from a project file shortens compilation time. Use caution when you remove files; you may need the data for troubleshooting.

Be aware that removing data here does not delete the entries for the removed data in the URL table. You must remove the data from the table using the Edit URL file function.

To remove obsolete data from a file:

1 Click **Remove Bin data.**

You see the Remove Binary Data dialog box, which displays the names of all binary files in the `bindata.c` file:

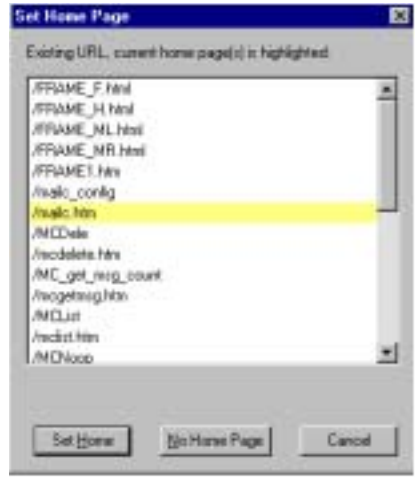


2 Select one or more files to delete from the list, and click **Remove.**

Setting or changing the project's home page

The home page is the page that appears in the browser when only the IP address of the server is typed in. The URL of a home page is always a forward slash (/).

Use **Home Page** to access the Set Home Page screen, from which you can add or change the home page of your project.



Setting the user and password of a URL

The HTTP Server supports the basic access authentication scheme in the HTTP protocol. Up to eight realms are allowed, each with its realm name, user name, and password. You need to set up a security table before you start the HTTP server.

A shell routine, `HSInitSecurityTable`, has already been implemented in the `url.c` file. To call `HSSetRealm` in `HSInitSecurityTable` to set up realm information, click **Set Security Realms** on the Edit URL file dialog box.

You see the Security Table dialog box:

To generate calls to `HSSetRealm` in `HSInitSecurityTable`, click **OK**. After that, you can assign each URL to one or more realms. **Add/Edit** on the Edit URL file screen lets you add or modify realms of a URL.

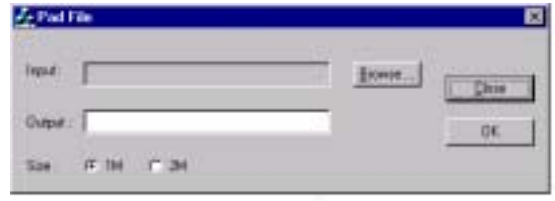
Pad File utility (Green Hills only)

Use the Pad File utility when you build an image for flash ROM. This utility extends a binary image to 1 or 2 Mbytes so you can line up the image evenly when you download it to the development board. After the image has been extended with the Pad File utility, the image is saved.

To pad a file:

1 Select **NETARM Windows-based Utility → PAD.**

You see this dialog box:



2 Click **Browse,** navigate to the file to pad, and select the file.

The same filename and location appear in the **Output** text input box. NetSilicon strongly recommends that you change either the directory, the filename in the Output text input box, or both. If you do not change these values, a message asks if you want to overwrite the file.

3 Click either **1M** or **2M.**

When the utility finishes padding the file, click **Close.**

FTP Download utility (Green Hills and GNU)

FTP Download lets you download new or updated software to flash ROM on the NET+Works development board. It supports 32-bit flash. This utility requires an FTP server to be running out of RAM on the NET+Works development board.

For information about creating a debug or boot ROM application running an FTP server out of RAM, see the *NET+OS Getting Started Guide* for the ICE you are using.

Preparing to use FTP Download

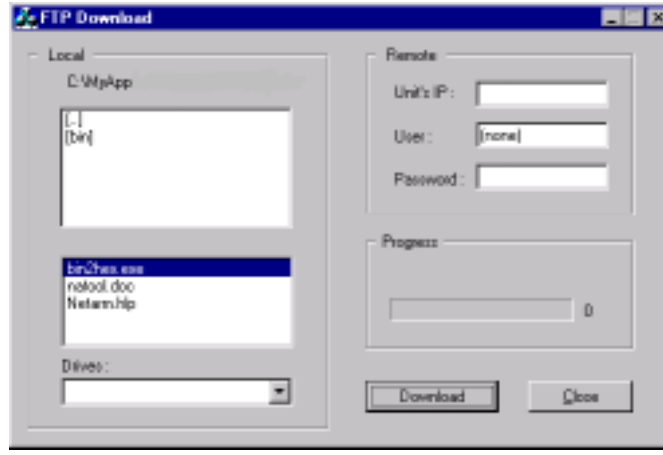
Before you use FTP Download, you need to establish communication with the board by running the FTP server application `naftpsvrapp`.

Using FTP Download

▶ To download software to flash ROM:

1 Select **NETARM Windows-based utility → FTP.**

You see this dialog box:



2 In the **Remote** section, type the IP address of the device for which you are downloading software.

You also may need to specify a username and password.

3 In the **Local** section, select the `rom.bin` file you want to download to the device.

4 In the **Drives** section,

5 Click **Download**.

When you are finished, click **Close**.

Adding FTP Download capability to an application

The `naftpapp` application is an example of how to create a ROM-based application that can update the contents of flash ROM. NetSilicon suggests that you use this application as a shell for any new application you develop that needs to update flash ROM.

A `naftprom` build file has two modules:

- The user application ROM-based program
- `ramimage.bld`, which consists of an image of the RAM-based FTP server application that lets you download a new image to the flash ROM

Because a ROM-based executable application cannot write to flash ROM itself, it must copy the FTP server (in this case, `ramimage`) into RAM and then execute the application from RAM. When the FTP server is running from RAM, it can communicate with an FTP client, such as the FTP download utility, and can download a new image into flash ROM.

But how does the first ROM-based image get written to flash ROM when there isn't an FTP server image (such as NAFTPROM) stored there already, or when the development board's ROM is completely empty? The `naftpapp` application can be used with the Debugger as a RAM-based FTP server, capable of downloading a new image into flash ROM. When the FTP server starts running, it waits until it receives the `programmer` file, which is the `rom.bin` file generated by the `naftprom` application. After the image is received, `naftpapp` stores the image into flash ROM, which is how the first ROM-based image gets into the development board.

Note that RAMSTART is defined to 0x100000 in the `naftprom.c` file. This is the location to which the execution pointer jumps when the program execution is transferred from ROM to RAM. This address is also specified in the `ramimage.lx`

file. If you want to change this address because of your memory requirements, you must match the definitions in both files.

Recovering from a flash download failure

If a failure (such as a power loss or network problem) occurs during an FTP download, or if garbage is accidentally downloaded to flash ROM, the development board may no longer be functional. In such a case, use the debugger to load `naftpapp` and reprogram flash ROM.

Application wizard (Green Hills only)

Using the Application wizard, you can customize a new application environment. The Application wizard provides the framework around which to build an application.

Although you can configure an application manually by editing the `appconf.h`, `debug.bld`, and `project.bld` files, the Application wizard saves you time. The application created is based on a minimal application template and is located in the directory `c:\netos4\src\apps`.

About the Application wizard

Because the Application wizard uses relative directories to specify paths, you must *not* change the pathname to your application. If you change the pathname, either cancel or edit the application name to recover the pathname. You can then copy the application to another directory. You may need to modify the pathnames, however, to build your application correctly.

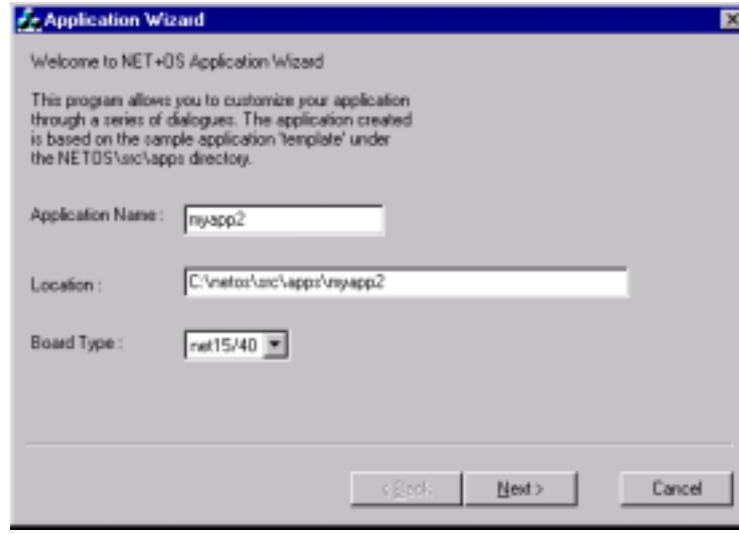
You must run the wizard under the path `\netos4\src\apps`.

Using the Application wizard

▶ To customize a new application environment:

1 Select **NETARM Windows-based utility → App Wizard**.

You see this page:

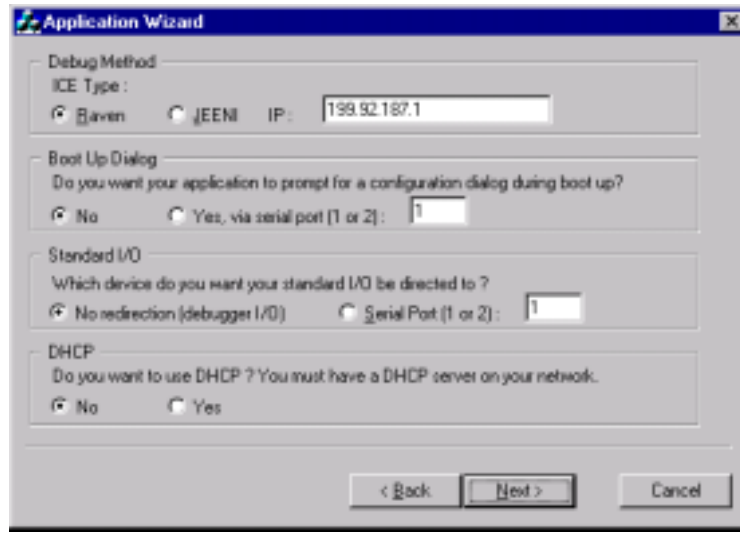


2 Make these selections:

To	Do this
Change the application's name	Type the name in the **Application Name** input box.
Use a different board type	Select the board you are using from the **Board Type** pull-down menu.

When you finish making selections, click **Next.**

You see this page:

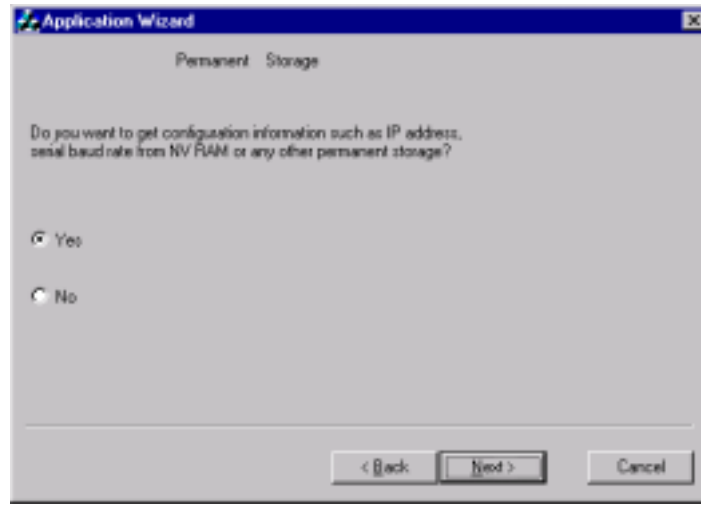


3 Make these selections:

In this section	Do this
Debug Method	1 Click the debug method to use. 2 Type the IP address of the device.
Boot Up Dialog	Specify whether the application will prompt for a configuration dialog box during boot-up. If you click **Yes, via Serial Port (1 or 2),** type the COM port to use.
Standard I/O	Select the device to which standard I/O will be directed. If you click **Serial Port (1 or 2),** type the COM port to use.
DHCP	Specify whether you want to use DHCP.

When you finish making selections, click **Next.**

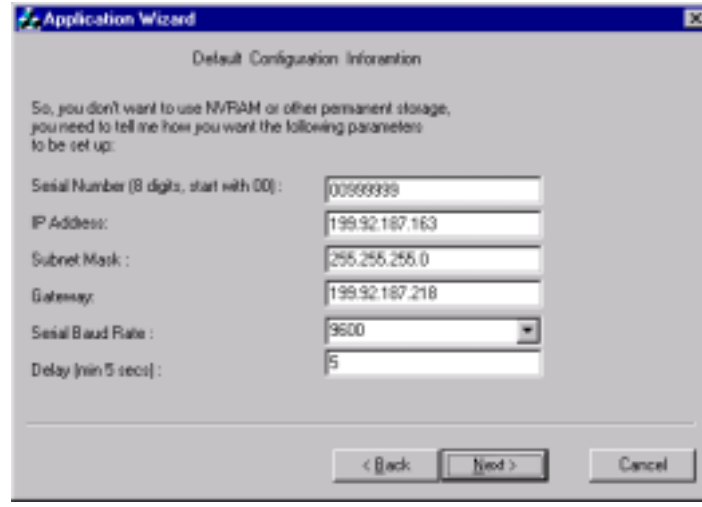
You see this page:



4 Do either of these steps:

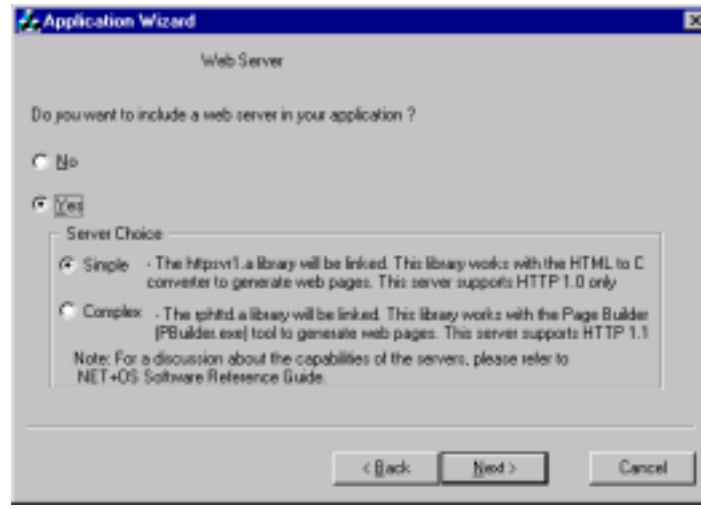
– Click **No,** and then click **Next.**

You see this page, which captures configuration information used for hardcoding into the device (as opposed to getting configuration information from NVRAM):



– Click **Yes,** and then click **Next.**

You see this page:

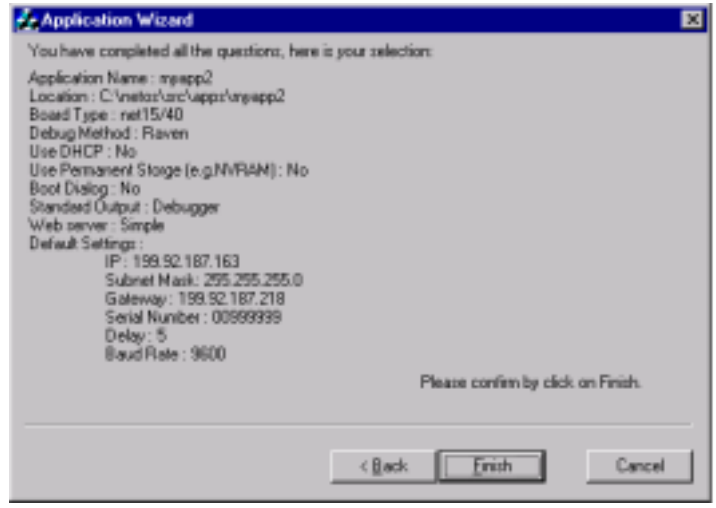


To specify whether to include a Web server in your application, click either **No** or **Yes.**

If you click **Yes,** click either **Simple (**standard Web server) or **Complex** (Advanced Web Server).

When you finish making selections, click **Next.**

You see this page:



5 Do one of these steps:

To	Click
Invoke and complete the application	**Finish**
Return to other pages to change your settings	**Back**
Cancel the wizard	**Cancel**

PBuilder Helper

PBuilder creates Web page stuff that is loaded onto the development board and used by the Web server. The PBuilder Helper opens a command line window you use to execute the PBuilder application.

This utility is described in detail in its own chapter. See Chapter 7, "Advanced Web Server PBuilder Utility."

Using MIBMAN

CHAPTER 9

This chapter describes the MIBMAN utility for implementing Management Information Bases (MIBs).

Overview

MIBMAN translates Small Network Management Protocol (SNMP) MIBs into C code that contains:

- Templates for the action routines that developers implement
- Management API declarations for MIB objects that correspond to management variables

Developers control which action routines and management variable declarations are generated through configuration files.

This section presents some basic terminology and concepts.

SNMP

SNMP defines a system for managing network devices and is implemented in different modules. SNMP consoles run on network workstations and provide the user interface. SNMP agents run on managed devices and handle communications with remote SNMP consoles. When a console sends a request to an agent, the agent decodes the request and calls subroutines in the managed device that handles the request. The subroutines are called *action routines* and are implemented by developers.

A MIB defines the view of a managed device that is presented to an SNMP console by the device's agent. The MIB defines a set of objects the console can read and write. These objects may either correspond to variables that exist on the device or be synthesized by the agent.

For example, a MIB might define one object that is the count of Ethernet packets received so far, and another object that indicates the health of the system. The count of Ethernet packets received is probably stored in a real variable. On the other hand, the health of the system may be determined by the agent on the fly by examining the state of several system variables and processes.

MIB objects that represent variables that exist are called*real objects*, and MIB objects that don't exist are called *virtual objects*.

Scalar MIB objects

A *scalar object* represents a single item with a simple type. If a scalar object is real, MIBMAN can generate most of the code needed to implement it.

MIBMAN can create a declaration for the variable in the management API and use generic action routines to read and write the variable. Then the developer has the application set an initial value for the variable and update it as necessary.

In some cases, generic action routines cannot be used. For example, a MIB object could represent the state of an LED, and special purpose action routines might be needed to turn the LED on and off when a user changes the value of the object. In this case, you can configure MIBMAN to generate templates for the action routines and a declaration for a management variable to represent it. If the object is virtual, you can use MIBMAN to generate templates for the action routines without generating a management variable declaration.

MIB tables

MIB objects can be arranged in tables. A *table* is a set of MIB objects repeated in multiple rows. Objects in a row are *columnar objects*. Columnar objects all have simple types, like scalar objects. An SNMP table is similar to an array of C structures, where each row represents a C structure, and each columnar object represents a field in the structure. Like scalar objects, tables can represent either real tables of information stored in memory, or virtual objects the agent creates on the fly.

Tables are read and written one columnar object at a time. When a columnar object is accessed, the object is identified with an object identifier (OID) that determines which columnar object in the row is being accessed, and an index that determines which row in the table is being accessed.

Although MIBs describe the elements that make up a table index, they do not describe how the index works. This information is explained in comments in the MIB and the RFCs that describe the MIB. The MIB developer implements the table's indexing scheme. Tables can have relationships with other tables; for example:

- One table can augment another table by defining additional columnar objects. In this case, the two tables have a one-to-one relationship.
- A table can be a different view of another table. For example, one table might consist of the same rows in another table sorted into a different order.
- Using complex indexing schemes, it is possible to make one table a subtable of another.

These relationships can cause the fate of rows in one table to affect rows in another table. For example, if one table augments another, when a row in one table is deleted, the corresponding row in the other table also must be deleted. These types of relationships are not described by the MIB, but in comments in the MIB or in RFCs that describe the MIB. The developer implements these relationships.

MIBMAN can create management variable declarations and action routine templates for MIB tables that represent real tables in memory. Developers must initialize the tables and update the data in them as necessary. Developers must also complete the action routines by coding the table's indexing scheme and fate relationships to other tables. If the table is virtual, MIBMAN can generate templates for its action routines.

Traps

A *trap* is a message an agent sends to a console when certain events occur. Although the contents of traps are defined inside MIBs, the MIBs do not specify the events that cause the agent to send the trap. These events are explained by comments in the MIB and in the RFC that describes the MIB. MIBMAN can be used to generate a set of data structures that applications use to generate and send a trap.

Action routines

Action routines are functions that are called by an SNMP agent to read or write MIB objects. MIBMAN generates templates of these functions that developers complete to implement the MIB. The agent passes parameters to the functions that identify which object is being accessed. The functions must perform the operation and return the result.

For more information about action routines, see the Fusion SNMP documenation.

Example (NAMIB)

`NAMIB` is an example program that demonstrates how to implement a MIB. The example MIB allows users to read and change the state of the LED on the board from a MIB browser. It also demonstrates how to implement a MIB with tables.

You can find `NAMIB` on your installation CD.

Basic operation

Converting an SNMP MIB into C code involves three steps:

1. A MIB compiler converts the MIB definition into an intermediate file.
2. A code generator converts the intermediate file into C code.

   You can use the SMICng MIB compiler to compile the MIBs. SMICng is an industry-standard compiler and is shipped with the SNMP agent.
3. MIBMAN converts the intermediate file generated by SMICng into three C source files.

   These source files contain the C variable definitions and action routine templates needed to implement the MIB.

SMICng

Because the authors of SNMP tried to design a format for SNMP MIB definitions that would be readable by both machines and people, MIB definitions are difficult to read for everyone and everything. The SMICng compiler converts a MIB definition into a form that is easier for software to read. For more information about this utility, see the Fusion SNMP documenation.

Here are the basic steps for using SMICng:

1. Strip off all extraneous text.

   MIB definitions are usually contained within text files (RFCs) that contain other text besides the MIB definition itself. If the MIB uses definitions from other MIBs, the text files that contain the definitions for these MIBs also must be cleaned up.

   This process is described in in the Fusion documentation.
2. Create an include file for SMICng that lists all the MIBs used by the one being processed. The final file included should be the MIB itself.
3. Run SMICng with this command line:

   `smicng -z -cm` *`yourMib.inc`* `>` *`yourMib.out`*

   where *`yourMib.inc`* is the name of the include file created in step 2, and *`yourMib.out`* is the name of the intermediate file the compiler creates.

MIBMAN

The MIBMAN utility operates on the intermediate files created by SMICng. MIBMAN accepts up to four parameters, as shown here:

`MIBMAN list-file configuration-directory output-directory var-filename`

where:

File	Description
list-file	File that contains a list of intermediate files generated by SMICng
configuration-directory	Directory where configuration files are located
output-directory	Directory where output files should be generated
var-filename	File that MIBMAN creates to contain a list of the variables defined by the MIB and their OIDs

The *list-file* parameter is required; the other three parameters are optional. If you do not specify the optional parameters, MIBMAN uses the current directory and does not generate a list of MIB variables.

The *list-file* must list the MIB files so that MIBs that have dependencies are listed after the MIBs that resolve the dependencies. For example, suppose RFC1213-MIB was being processed. This MIB depends on items defined in RFC1155-MIB and RFC1212-MIB. So, a list file for RFC1213-MIB would be:

```
RFC1155-MIB.MIB
RFC1212-MIB.MIB
RFC1213-MIB.MIB
```

Generated files

MIBMAN generates two C files and a header file (.h) for each MIB it processes that defines variables or traps. By default, filenames are determined by the name of the MIB module. The MIB module name is converted into legal Windows filenames by changing hyphens and periods in its name into underscores and appending the extensions .c and .h. The suffix *Action* is appended to the template filename. If the module name starts with a digit, an underscore is added to the beginning of the filename.

One C file (called the *definition file*) contains the definitions for variables and structures used by the management API and the SNMP agent, including:

- An array of `manVarType` elements used to create the variables in the management API database
- An array of `struct variable` elements used to register the variables with the SNMP agent
- Information about SNMP traps that is needed to implement the traps

The other C file (the template file) contains templates for action routines that are needed to implement the MIB.

The header file contains declarations for functions and variables defined in the two C files, including declarations for:

- The `manVarType` and `struct variable` arrays defined in the definition file
- The action routine templates in the action file that read variables
- The constants needed to implement the action routines

By default, MIBMAN generates the C code by:

- Creating management variables to to represent all scalar objects
- Using generic action routines to access all scalar objects
- Creating management table variables to represent all MIB tables
- Creating templates or action routines that read and write the tables

None of the management variables is protected by semaphores. Indexing for management table variables is left undefined. These default settings can be changed through a configuration file.

Data types

SNMP data types are represented by similar management variable data types:

These SNMP data types	Are represented as
Textual conventions, octet strings, and opaque objects	MAN_OCTET_STRING_TYPE variables
INTEGER and Integer32 data types	INT32 types
Unsigned32, Gauge, Gauge32, Counter, Counter32, and TimeTicks	WORD32 types

These SNMP data types	Are represented as
Counter64 data types	WORD64 type
OBJECT IDENTIFER	Arrays of 129 WORD32 elements. The length of the OID is encoded in the last element of the array.
IPAddress and NetworkAddress types	An array of four WORD8 elements
BITS	Octet strings in which each element contains 8 bits. SNMP bit label 0 is the least significant bit in the first byte of the string. The number of bit labels defined in the MIB determines the string's length.

Tables

Tables are registered with the management API using the `MAN_TABLE_TYPE` data type. Individual rows are represented as C structures where each columnar element is represented as a field in the structure.

In this example, a table is defined with two integer objects and a `DisplayString` object. A row in the table is represented as a C structure with two `INT32` fields and an `MAN_OCTET_STRING_TYPE` field.

```
ifTable OBJECT-TYPE
          SYNTAX  SEQUENCE OF IfEntry
          ACCESS  not-accessible
          STATUS  mandatory
          DESCRIPTION
                      "A list of interface entries."
              ::= { interfaces 2 }

ifEntry OBJECT-TYPE
              SYNTAX  IfEntry
              ACCESS  not-accessible
              STATUS  mandatory
              DESCRIPTION
                      "An interface entry."
              INDEX   { ifIndex }
              ::= { ifTable 1 }
```

```
IfEntry ::=
              SEQUENCE {
                  ifIndex
                      INTEGER,
                  ifDescr
                      DisplayString,
                  ifType
                      INTEGER,
              }
ifIndex OBJECT-TYPE
              SYNTAX  INTEGER
              ACCESS  read-only
              STATUS  mandatory
              DESCRIPTION
                      "A unique value for each interface."
              ::= { ifEntry 1 }

ifDescr OBJECT-TYPE
              SYNTAX  DisplayString (SIZE (0..255))
              ACCESS  read-only
              STATUS  mandatory
              DESCRIPTION
                      "A textual string containing a description."
              ::= { ifEntry 2 }

ifType OBJECT-TYPE
              SYNTAX  INTEGER
              ACCESS  read-only
              STATUS  mandatory
              DESCRIPTION
                      "Identifies the type of interface."
              ::= { ifEntry 3 }
```

MIBMAN creates a definition for a structure that represents individual rows in the table:

```
typedef struct
{
          INT32 ifIndex;
          MAN_OCTET_STRING_TYPE ifDescr;
          INT32 ifType;
} IfEntryType;
```

Integration with the management API

MIBMAN creates an array of `manVarType` objects with one element for each scalar object and table defined in the MIB. Developers can suppress entries for virtual objects.

By default, MIBMAN sets the fields in the `manVarType` structure for each MIB variable, as shown here:

Field	Description
id	Set to the objects's OID expressed as a string
varPointer	Set to NULL by default
isFunction	Set to 0 by default
size	Set to the size of the variable in bytes
type	Set to indicate the variable's type
dimensions	Set to NULL
numberDimensions	Set to 0
semaphores	Set to NULL by default
numberSemaphores	Set to 0 by default
rangeFn	Loaded with the address of the snmpAgentRangeChecker function if the variable has a range
rangeInfo	Loaded with the address of an snmpAgentRangeType structure that describes it, if the variable has a range; otherwise, it is set to NULL
tableInfo	Set to NULL if the object is a scalar. If the object is a table, this field points to a manTableInfoType structure with information about it.
callbackFn	Set to NULL

Implementing traps

Traps are messages sent to SNMP consoles and other SNMP agents when a predefined event occurs. The MIB definition specifies the content of trap messages, but not the events that trigger them. To support traps, the developer must implement code that detects when a trap event occurs and calls functions in the SNMP agent to create a trap message and send it.

MIBMAN creates an `snmpAgentTrapType` structure for each trap defined in the MIB. The structure contains information about the trap that is needed to construct the trap message. The `snmpAgentTrapType`structure is defined in this way:

```
typedef struct
{
          char *name;
          char *id;
          int trapNumber;
          int trapType;
          int variableCount;
          snmpAgentTrapVarType *varList;
} snmpAgentTrapType;
```

where:

Field	Description
id	Loaded with the trap's OID
trapNumber	Set to the trap number
trapType	Set to `TRAP_TYPE` if the trap is an SMI version 1 trap, or `NOTIFICATION_TYPE` if the trap is an SMI version 2 notification
variableCount	Loaded with the number of variables in the trap
varList	Pointer to a list of the variables

The trap variables are listed in an array of `snmpAgentTrapVarType` structures. The `snmpAgentTrapVarType` structure is defined in this way:

```
typedef struct
{
    char *id;
    char *oid;
    int type;
} snmpAgentTrapVarType;
```

where:

Field	Description
id	The variable's ID (usually its OID)
oid	The variable's OID
type	A constant defined in ASN1.H that indicates the variable's data type

MIBMAN configuration file

You can override some default values used by MIBMAN through a configuration file. The configuration file's name is determined by the name of the MIB being processed. The MIB module name is converted into legal Windows filenames by changing any hyphens and periods in the module name into underscores and appending the extension .config. If the module name starts with a digit, an underscore is prefixed to the filename.

For example, suppose MIB-II is being compiled. The MIB definition begins with this statement:

```
RFC1213-MIB DEFINITIONS::= BEGIN
```

MIBMAN creates the configuration file name by converting the MIB name RFC1213-MIB into the filename RFC1213_MIB.config.

If the configuration file does not exist, the default values described above are used. Otherwise, MIBMAN opens and reads the file. The file should consist of one or more configuration statements. Configuration statements take one of two forms:

- *keyword value*
- *keyword OID = value*

The configuration file also can have comments, which start with a semicolon. Any text after a semicolon is ignored.

Controlling the names of generated files

By default, MIBMAN bases the names of the .c and .h files on the MIB's name . This can be overridden. To set the name of the .c and .h files, use these statements:

- Cfilename *c-filename*
- Hfilename *h-filename*
- Actionfilename *action-filename*

where *c-filename*, *h-filename*, and *action-filename* are replaced with your filenames.

Overwriting files

The definitions C file and the header file generated for a MIB are overwritten by MIBMAN every time it is run. However, the action C file contains template code that you must edit. MIBMAN does not overwrite this file; you must delete or rename the action file for MIBMAN to create a new one.

Controlling the comments in the C and header files

When MIBMAN generates the C and header files, it generates comments at the top of the file. You can set some of the information in these comments through configuration options.

The options take this form:

Option	Description
Author *author*	Sets the author listed in the comments of the C and header files.
ModuleName *modulename*	Sets description information. Any number of description options can be specified.
Description *description*	Lets you keep an edit history. You specify an edit ID and text to go along with it.
Edit *EditId=text*	The edit ID can be up to 14 characters. The date of the change is a good ID. The *text* component contains a description of the edit. If the description is longer than one line, it can be extended to any number of lines by using the same edit ID. All the description text is inserted into the file.

Here is an example:

```
Author Harry Hu
ModuleName Management Information Base for Network Management
Description This defines second version of the Management Information
Description Base (MIB-II) for use with network management protocols in
Description TCP/IP based internets.
Edit 12/12/99-HH=Original code generated.
Edit 02/20/00-HH=Increased size of IP connect table, added semaphores
Edit 02/20/00-HH=to protect access to all variables.
Edit 03/12/00-HH=Updated with changes to IP stack.
```

In this example:

- The *author* field in the C and header files is set to Harry Hu.
- The module name is set to *Management Information Base for Network Management*.
- The *description* field is divided into the three lines of text.
- Three edits are specified with IDs and descriptions.

Controlling management variables

Suppressing management variables

Use DontCreateVariable to prevent MIBMAN from generating a management variable declaration for a MIB object.

```
DontCreateVariable oid
```

This option is useful if it is a virtual object or if the management variable for it has already been defined. MIBMAN always generates action routines for the object when this option is specified.

Setting global variable prefixes

```
MibVariablePrefix prefix
AgentVariablePrefix prefix
```

The MibVariablePrefix and AgentVariablePrefix options are used to specify prefixes for global variables that MIBMAN creates. This can help avoid collisions with other global variables in the application. MibVariablePrefix is used for variables that are related to MIB items. AgentVariablePrefix is used for global variables that are not tied to a specific MIB.

Setting the management ID

```
SetVariableIdentifier oid = identifier
```

The SetVariableIdentifier option is used to specify the management API's identifier for the variable. By default, MIBMAN uses the object's OID represented as a character string. This option can be used to specify a different identifier. For example, this statement sets the identifier of sysDescr to SystemDescription:

```
SetVariableIdentifier 1.3.6.1.2.1.1.1 = SystemDescription
```

Setting index information

```
SetIndexFunction oid = function
SetIndexInfo oid = info
```

Management table variables may use complex indexing systems. MIBMAN creates the variable without an index system by default, which means rows must be read using a simple numeric index. If this is not sufficient, `SetIndexFunction` and `SetIndexInfo` can be used to specify a complex indexing system. The `SetIndexFunction` option sets the name of a function that the management API uses to index into the table. The `SetIndexInfo` option sets the name of the buffer passed to the index function.

For example, these statements set the indexing function to tableIndexer, and the index information to `tableIndexInfo`:

```
SetIndexFunction 1.3.6.1.45.65 = tableIndexer
SetIndexInfo 1.3.6.1.45.65 = tableIndexInfo
```

Setting semaphores

The `SetGlobalSemaphore` and `SetSemaphore` options can be used to specify that some or all the variables are to be protected by semaphores.

```
SetGlobalSemaphore semaphore
SetSemaphore oid = semaphore
```

The `SetGlobalSemaphore` sets a semaphore that is used to protect all variables in the MIB. `SetSemaphore` specifies a semaphore that protects a specific MIB.

Configuration files can have any number of `GlobalSemaphore` and `Semaphore` statements; each specifies a single semaphore to use. When MIBMAN generates a `manVarType` entry, it loads the semaphores field with a pointer to an array that contains all the semaphores that have been specified both globally and for the specific variable.

The `Semaphore` statement does not override the `GlobalSemaphore` statement. Instead, it specifies semaphores that will be used with the variable in addition to those used for all variables.

Generating action routines

MIBMAN automatically generates action routines for all tables and for virtual scalar objects. Normally, MIBMAN does not generate action routines for real scalar objects because these objects can be handled by the agent with generic action routines. In some cases, it is desirable to create custom action routines for real scalar objects when reading or writing them is supposed to cause side effects.

Use `GenerateWriteActionRoutine` and `GenerateReadActionRoutine` to generate templates for these routines. These options take this form:

```
GenerateWriteActionRoutine oid
GenerateReadActionRoutine oid
```

For example, this option would cause MIBMAN to generate a template action routine to write the value of a scalar object whose OID is 1.3.6.1.4.1.901.999.1.1.1:

```
GenerateWriteActionRoutine 1.3.6.1.4.1.901.999.1.1.1
```

Setting accessor functions

Management variables are usually controlled by the management API. However, it is possible to create variables that are accessed through the API but implemented in application code. This is done by specifying an accessor function for the variable when it is registered with the API. If an accessor function is specified, the management does not allocate any memory for the variable, and calls the accessor function when an application tries access the variable. The accessor function is responsible for implement reads and writes to the variable.

Use `SetAccessorFunction` to specify that the management variable created for an object should have an accessor function. `SetAccessorFunction` takes this form:

```
SetAccessorFunction oid = function
```

For example, use this option to assign the accessor function `sysDescrFunction` to the sysDescr item in MIB-II:

```
AccessorFunction 1.3.6.1.2.1.1.1=sysDescrFunction
```

Include files

Include files can be used to declare externally defined variables that are used in the definition C file. For example, if a semaphore is used, it must be defined in an external module. The `Include` statement specifies the name of a file to include in the definitions C file and takes this format:

```
Include filename>
```

where *`filename`* specifies the name of the file to include. Any number of include files can be specified.

Controlling the names of constants

MIBMAN defines constants in the header file to represent variable identifiers. By default, the names of the constants are determined by concatenating the name of the MIB and the name of the variable with an underscore character between them.

For example, MIBMAN defines a constant named `RFC1213_MIB_sysDescr` to represent the ID of the variable `sysDescr in RFC1213_MIB`.

You can use the `IdentifierPrefix` option, which takes this form, to set the prefix for identifiers:

`IdentifierPrefix` *`prefix`*

where *`prefix`* specifies the prefix to use. For example, using the option `IdentifierPrefix MIBII_` in the configuration file for `RFC1213_MIB` would change the name of `RFC1213_MIB_sysDescr` to `MIBII_sysDescr` (and all other constants too).

Final integration

After using MIBMAN to generate the C and header fields, do these:

1. Edit the code in the template files and implement the missing code. The missing code is marked with "To Do" comments.
2. Code any index and accessor functions needed by the MIB variables.
3. Create and initialize and semaphores that are used to protect the variables before the variables are registered with the management API.
4. Provide code that calls the `snmpRegisterManagementVariables`, which:
   - Registers the management variables created for the MIB
   - Sets default values for scalar objects (if specified in the MIB)
   - Registers the MIB objects with the SNMP agent
5. Provide code to set initial values of MIB objects that do not have default values defined in the MIB.
6. Provide code to implement traps.

   The code must detect when an event occurs, build a trap message, and send it. MIBMAN will have created data structures in the C file that contains some of the information needed to generate and send trap messages.

Writing action routines

Action routines for scalar objects

If a scalar object is represented by a real variable in the management database, it is usually not necessary to write an action routine for it. MIBMAN defines the management variable for the object, and the agent uses default action routines to read and write the variable.

If a scalar object cannot be represented by a variable in the management database, the developer must write actions routines to read and write the object. This is controlled by the `DontCreateVariable` configuration option. In this case, MIBMAN generates templates for the read and write action routines. Developers must finish these templates to perform the operations.

For example, suppose a MIB variable represents the state of an LED:

- When the variable is read, the LED hardware should be interrogated to determine the value to return.
- When the variable is written, the LED should be turned on or off.

MIBMAN generate a template read action routine similar to this:

```
void *mib_LEDRead (struct variable *vp, oid *name, int *length,
int isGet, int *varLen, setMethod *setVar)
{
    void *resultBuffer = NULL;

    if (!scalar_handler(vp, name, length, isGet, varLen))
    {
        return NULL;
    }

/*
* To Do:  Read LED(1.3.6.1.4.1.901.999.1.1.1) into a persistent
*         buffer, set resultBuffer to point to it, and set
*         *varLen to the value's length.
*/

    *setVar = vp->writeFn;
    return resultBuffer;
}
```

This action routine must be edited to determine whether the LED is on or off and to return a pointer to an integer value (1 or 0) to the SNMP agent. The data must be stored in a persistent buffer that is statically allocated or is not freed until after the SNMP agent sends the reply to the console.

Here is an edited version that does this:

```
void *mib_LEDRead (struct variable *vp, oid *name, int *length,
            int isGet, int *varLen, setMethod *setVar)
{
    static int LEDState;
    void *resultBuffer = &LEDState;

    if (!scalar_handler(vp, name, length, isGet, varLen))
    {
        return NULL;
    }

    if (LedIsOn())
    {
        LEDState = 1;
    }
    else
    {
        LEDState = 0;
    }

    *setVar = vp->writeFn;
    return resultBuffer;
}
```

Read action routines also must return a pointer to the write action routine. The template created by MIBMAN already contains code to do this.

MIBMAN creates a template write action routine similar to this:

```
int agent_sysContactWrite (int actionCode, struct varBind *info)
{
    int result = SNMP_ERR_NOERROR;

    switch (actionCode)
    {
        case SNMP_SET_RESERVE:
```

```
                /*
                 * Ignore this action code unless you need to
                 * allocate memory.
                 */
                 break;
            case SNMP_SET_COMMIT:
                /*
                 * Ignore this action code.
                 */
                 break;
            case SNMP_SET_ACTION:
                /*
                 * To Do:  Write code to copy info->setTo into
                 *         the object.
                 */
                 break;
            case SNMP_SET_FREE:
                /*
                 * To Do:  Free any buffers you allocated.
                 */
                 break;
            case SNMP_SET_UNDO:
                /*
                 * To Do:  Write code to copy info->val into
                 *         the object.
                 */
                 break;
        }

        return result;
    }
```

The SNMP agent breaks the write process into four phases and an undo operation. These are indicated by the `actionCode` parametr. For scalar objects, the only phases that are important are the `SNMP_SET_ACTION` phase where the new value is written and the `SNMP_SET_UNDO` phase where the original value is restored.

The agent provides the value to write in both the `SNMP_SET_ACTION` and `SNMP_SET_UNDO` operations. In the `SNMP_SET_ACTION` phase, the value is stored in `info->setTo`; in the `SNMP_SET_UNDO` phase, the value is stored in `info->val`.

This example shows how the template could be edited to change the state of the LED:

```
int agent_sysContactWrite (int actionCode, struct varBind *info)
{
    int result = SNMP_ERR_NOERROR;

    switch (actionCode)
    {
        case SNMP_SET_ACTION:
            if (info->setTo.intVal == 0)
            {
                TurnLedOff();
            }
            else
            {
                TurnLedOn();
            }
             break;
        case SNMP_SET_UNDO:
            if (info->val.intVal == 0)
            {
                TurnLedOff();
            }
            else
            {
                TurnLedOn();
            }
            break;
        default:
            break;
    }

    return result;
}
```

You may need to create action routines for scalar values that are represented by management variables.

For example, suppose a MIB object represents the speed of a motor. When the value is written, not only does the management variable need to be changed, but the speed of the motor also should be adjusted. In this case, the `GenerateWriteActionRoutine` can be used to make MIBMAN generate a write action routine for the variable, as shown here:

```
int mib_MotorSpeedWrite (int actionCode, struct varBind *info)
{
    int result = SNMP_ERR_NOERROR;

    if (actionCode == SNMP_SET_ACTION)
    {
        result = snmpWriteObject(info->vp, &info->setTo,
          info->setToLen);
    }
    else if (actionCode == SNMP_SET_UNDO)
    {
        result = snmpWriteObject(info->vp, &info->val,
                  info->valLen);
    }

    return result;
}
```

The write action routine already updates the management variable. It must be edited to set the motor speed to the new value, as shown here:

```
int mib_MotorSpeedWrite (int actionCode, struct varBind *info)
{
    int result = SNMP_ERR_NOERROR;
    int motorSpeed;

    if (actionCode == SNMP_SET_ACTION)
    {
        result = snmpWriteObject(info->vp, &info->setTo, info-
>setToLen);
        motorSpeed = info->setTo.intVal;
    }
    else if (actionCode == SNMP_SET_UNDO)
```

```
    {
        result = snmpWriteObject(info->vp, &info->val, info->valLen);
        motorSpeed = info->val.intVal;
    }

    if (   (result == SNMP_ERR_NOERROR)
        && ((actionCode == SNMP_SET_UNDO) || (actionCode ==
SNMP_SET_ACTION)))
    {
        setMotorSpeed(motorSpeed);
    }

    return result;
}
```

Action routines for tables

SNMP tables are not fully defined by the MIB. Additional information is included in comments in the MIB's RFC:

- **How the table's indexing scheme works.** The console supplies a table index whenever it accesses a columnar object. The table index is used to identify which row in the table is being read or written. Although the MIB describes what type of index values are passed, it does not describe how they are to be used. This is described in comments.

  Some table indexing systems are very complex and create relationships with other tables. The action routines that MIBMAN creates for tables must be modified to implement the indexing scheme.

- **How rows are inserted and deleted.** If the console has the ability to add and delete elements in the table, the algorithm for doing this is also described in the MIB's RFC.

  RFC-1903 describes the recommended approach for doing this, but some MIBs may use different methods.

Because this information is not included in the MIB, MIBMAN always generates action routine templates for table objects the developer has to complete. The same read and write action routines are used by all columnar objects in a table.

A typical read action routine template for a table looks like this.

```
void *someTableRead (struct variable *vp, oid *name, int *length,
            int isGet, int *varLen, setMethod *setVar)
{
    void *resultBuffer = NULL;
    manVarType *manInfo = snmpGetVariableInfo(vp);
    MAN_ERROR_TYPE ccode;
    snmpIndexType *snmpIndex = snmpExtractIndices(vp, name,
*length, IS_READ, 13);

    *varLen = 0;

    if (snmpIndex != NULL)
    {
        manTableIndexType manIndex; /* index for management API */
        someTableType *row; /* storage for one row from table*/
        row = (someTableType *) malloc(sizeof(someTableType));

        if (row != NULL)
        {
            memset(row, 0, sizeof(someTableType));

            manIndex.wantExact = isGet;
            if (snmpIndex->isNullIndex)
            {
                manIndex.numericIndex = 0;
                manIndex.snmpIndex = NULL;
            }
            else
            {
           /*
                * The raw SNMP indices are stored in snmpIndex. The
                * algorithm for using these indices should be
                * described somewhere in the MIB's RFC.
```

```
                 *
                 * To Do: Write code to initialize manIndex. For a
                 * GET manIndex must be the exact index of the row
                 * to read. It must be one past it for a GET-NEXT.
            */
             }
             ccode = snmpReadRow(manInfo, &manIndex, row);
             if (ccode == MAN_SUCCESS)
             {
                 resultBuffer = snmpExtractField(vp, row, varLen
                                NULL);

                 /*
                  * To Do: Update the index values in snmpIndex to
                 * reflect the actual index the row is at. This will
                 * be encoded into the name parameter by the call to
                  * snmpEncodeIndices.
                  */

                 memcpy(name, vp->name, vp->namelen *
sizeof(WORD32));
                 *length = snmpEncodeIndices(vp, name, snmpIndex);

                 if (snmpFreeBufferLater(resultBuffer) !=
SNMP_ERR_NOERROR)
                 {
                     free(resultBuffer);
                     resultBuffer = NULL;
                     *varLen = 0;
                 }
             }
             snmpFreeOctetStringBuffers(manInfo, row);
             free(row);
        }
```

```
            else /* if unable to allocate memory for row buffer */
            {
                resultBuffer = NULL;
                *varLen = 0;
            }
            snmpFreeIndices(snmpIndex);
        }

        *setVar = vp->writeFn;

        return resultBuffer;
}
```

The template extracts the table index from the OID and puts it into the local variable `snmpIndex`. The developer must implement the table's indexing algorithm to convert the index into an index that can be used by the management API to look up a row. This is indicated by a comment in the template with a "To Do" note.

After the index has been converted into a form that can be used by the management API, the table is read into the row buffer. When MIBMAN processes the MIB, it creates a C structure that represents rows in the table. The row variable is defined to be one of these structures. Once the row is read, the field in the structure that represents the columnar object being read is extracted from it and copied into a buffer that is returned to the agent.

If the read was a `GET-NEXT` operation, the index specified by the console might not be the actual index of the row that was read. In this case, the function must also update the OID value passed to it with the new index. The action function must update `snmpIndex` to reflect the actual index of the row, and then encode it into the OID with the `snmpEncodeIndices`function.

Writing to table elements is more complicated. The SNMP agent breaks up the write operation into five phases, and the phases are indicated by the value of the `actionCode` parameter passed to the write action function.

These phases are:

Phase	Description
SNMP_SET_RESERVE	During this phase, the action routine allocates memory or other resources needed to perform the write operation.
SNMP_SET_COMMIT	The action routine should copy the value in info->setTo to the buffers allocated in the SNMP_SET_RESERVE phase.
SNMP_SET_ACTION	The action routine should write the variable at this point.
SNMP_SET_FREE	The action routine releases the buffer or other resources previously allocated.
SNMP_SET_UNDO	This phase occurs only if an error is detected. The action routine should restore the variable's original value. The value will be in info->val.

This code is an action routine template that writes into a table:

```
int someTableWrite (int actionCode, struct varBind *info)
{
    int result = SNMP_ERR_NOERROR;
    snmpIndexType *snmpIndex = snmpExtractIndices(info->vp,
          info->oid, info->oidLen, IS_WRITE, 13);
    manVarType *manInfo = snmpGetVariableInfo(info->vp);
    manTableIndexType oldIndex;
    static manTableIndexType newIndex;
    static someTableType *row = NULL;
    static int isInsert = 0, didWrite = 0, startedUndo = 0;
    int fieldCode = snmpGetFieldCode(info->vp);
    static int lastActionCode = -1;
    static WORD32 fieldsCommitted[1];
    static WORD32 fieldsUndone[1];
    MAN_ERROR_TYPE ccode;
```

```
        /*
         * The raw SNMP indices are stored in snmpIndex.  The
         * algorithm for using these indices should be described
         * somewhere in the MIB's RFC.
         *
         * To Do:  Write code to initialize oldIndex.
         */
        oldIndex.wantExact = 1; /* always use exact index for writes */

        switch (actionCode)
        {
            case SNMP_SET_RESERVE:
                if (row == NULL)
                {
                    row = malloc(sizeof(someTableType));
                   memset(fieldsCommitted, 0, sizeof(fieldsCommitted));
                    memset(fieldsUndone, 0, sizeof(fieldsUndone));
                    isInsert = 0;
                    didWrite = 0;
                    startedUndo = 0;
                    result = snmpInitRow(manInfo, &oldIndex,
                           sizeof(someTableType), &isInsert, &row);
                }
                if (result == SNMP_ERR_NOERROR)
                {
                    result = snmpAllocateFieldBuffer(actionCode,
                             info, row);
                }
                break;
            case SNMP_SET_COMMIT:
                result = snmpSetField(actionCode, info, row);
                fieldsCommitted[fieldCode / 32] |= 1 << (fieldCode &
0x1f);
```

```
            break;
        case SNMP_SET_ACTION:
            if (lastActionCode == SNMP_SET_COMMIT)
            {
/*
          * To Do: The array fieldsCommitted will indicate which
          * fields the console has provided values for. Set default
          * values for any fields that are missing, and verify that
          * the row contains valid data and can be written into
          * the table.
*/
                if (isInsert)
                {
                   memcpy (&newIndex, &oldIndex, sizeof(newIndex));
                   ccode = manInsertSnmpRow(manInfo->id, &newIndex,
                           row, MAN_TIMEOUT_FOREVER);
                }
                else
                {
                    /*
                     * To Do: Initialize newIndex to indicate the
                     * row's new position in the table.
                     */
                    newIndex.wantExact = 1;
ccode = manSetSnmpRow(manInfo->id, &oldIndex, &newIndex, row,
MAN_TIMEOUT_FOREVER);
                }
                if (ccode == MAN_SUCCESS)
                {
                    didWrite = 1;
                }
                result = snmpErrorLookup[ccode];
            }
            break;
```

```
            case SNMP_SET_FREE:
                if (row != NULL)
                {
                    /*
                    * Free our buffers.  This is only done on the first
FREE call.
                     */
                    snmpFreeOctetStringBuffers(manInfo, row);
                    free(row);
                    row = NULL;
                }
                break;
            case SNMP_SET_UNDO:
                if ((row != NULL) && (!startedUndo))
                {
                    snmpFreeOctetStringBuffers(manInfo, row);
                    free(row);
                    row = NULL;
                }
                if (didWrite)
                {
                    if (isInsert)
                    {
                      ccode = manDeleteSnmpRow(manInfo->id, &newIndex,
MAN_TIMEOUT_FOREVER);
                        if (ccode != MAN_SUCCESS)
                        {
                            result = SNMP_ERR_UNDOFAILED;
                        }
                        didWrite = FALSE;
                    }
                    else
                    {
                        if (!startedUndo)
```

```
                    {
                        result = snmpInitRow(manInfo, &newIndex,
sizeof(someTableType),
NULL, &row);
                        startedUndo = 1;
                    }
                    if (result == SNMP_ERR_NOERROR)
                    {
                      result = snmpAllocateFieldBuffer(actionCode,
info, row);
                    }
                    if (result != SNMP_ERR_NOERROR)
                    {
                        snmpFreeOctetStringBuffers(manInfo, row);
                        free(row);
                        row = NULL;
                        result = SNMP_ERR_UNDOFAILED;
                        break;
                    }
                    snmpSetField(actionCode, info, row);
                   fieldsUndone[fieldCode / 32] |= 1 << (fieldCode
& 0x1f);
                    if (memcmp(fieldsCommitted, fieldsUndone,
sizeof(fieldsUndone)) == 0)
                    {
                     ccode = manSetSnmpRow(manInfo->id, &newIndex,
&oldIndex, row,
MAN_TIMEOUT_FOREVER);
                        if (ccode != MAN_SUCCESS)
                        {
                            result = SNMP_ERR_UNDOFAILED;
                        }
                        snmpFreeOctetStringBuffers(manInfo, row);
                        free(row);
                        row = NULL;
```

```
                        startedUndo = 0;
                        didWrite = 0;
                    }
                }
            }
            break;
    }
    lastActionCode = actionCode;

    snmpFreeIndices(snmpIndex);
    return result;
}
```

Often, a single request from the console writes several columnar objects in a table. For example, the console must specify all the columnar objects to insert a new row. During each phase, the write action routine is called for each columnar object specified in the request. If a table has five columnar objects, during a write, the action routine is called five times for the `SNMP_SET_RESERVE` phase, then five times for the `SNMP_SET_COMMIT` phase, and so on.

The template routines MIBMAN generates handle most of the work for typical tables.

Routine	Description
`SNMP_SET_RESERVE`	Allocates a buffer for the row. If the row contains octet strings, the template code allocates buffers large enough to hold the new values the console is setting. The template code also attempts to read the row at the index the console specifies. If successful, the current values for the row is copied into the row buffer and isInsert is set to FALSE. If there is no row at the index, the row buffer is zeroed and isInsert is set to TRUE.
`SNMP_SET_COMMIT`	Copies the values for the columnar objects set by the console into the row buffer.
`SNMP_SET_ACTION`	Inserts the new row into the table if isInsert is set; otherwise, the existing row at the specified index is updated. Developers need to: ■ Add code to set default values for fields that are not set during an insert (including index fields). ■ Verify that all required fields have been set by the console. ■ Verify that the row contains valid data.

Routine	Description
SNMP_SET_FREE	Frees all allocated buffers.
SNMP_SET_UNDO	Either deletes the row, if one was inserted, or restores the row's original value.

Developers must implement this code in table write action routines:

- As with the read action routine, developers must implement the table's index scheme to encode the SNMP style index oldIndex. If the write can change the index of the row, the developer must also update newIndex with the new index values.
- The console does not necessarily provide values for all the columnar objects in a row. Developers must implement code to set default values for any the console doesn't provide and verify that all the required values have been provided.
- Developers must provide code to verify that the values in the row are valid.
- If rows can be deleted, developers must implement the code to do this. Usually, this is done by changing one of the columnar objects in the row to indicate that the row is no longer valid. The action routine should detect when this occurs and delete the row from the table, rather than just updating the field value.

SNMP OID and string index values

OIDs and octet string indexes can be encoded in two different ways. When snmpExtractIndices() encodes strings and OIDs into snmpIndexType structures, it always encodes the length of strings into the length field of the MAN_OCTET_STRING_TYPE that holds the strings, and the length of the OIDs into the oidLength field of snmpIndexComponent structure. However, non-implicit OID and octet string indexes have an additional length field at the beginning of the value. These length fields are considered part of the value of the string or the OID, and so change the way the values are ordered.

For example, if May and June were encoded as non-implicit octet string index values, May would come before June because May has a length of 3, and June has a length of 4. They could be represented as <3>May, and <4>June. Because 3 comes before 4, <3>May comes before <4>June. However, when the same strings are encoded as implicit octet strings, the length byte is dropped so June comes before May since J comes before M.

When the `snmpExtractIndices` function is used to decode octet string and OID index values, it encodes implicit values without a length field, but encodes non-implicit values with the length of the string in the first element of the string. This sets up the index values so they can be used for compariso as described in the SNMP RFCs. The index types are set to either:

- `SNMP_STRING_INDEX` and `SNMP_OID_INDEX` if the values are implicit
- `SNMP_STRING_INDEX_WITH_LEN` and `SNMP_OID_INDEX_WITH_LEN` if they are not implicit.

Sometimes MIB designers are not aware of the difference between implicit and non-implicit index values. They may not expect nonimplicit indexes to be affected by their length. Therefore, it is important to carefully read the MIB's RFC to determine what the MIB designer intended.

Using the Advanced Web Server PBuilder Utility

CHAPTER 7

This chapter describes the PBuilder utility, converts HTML Web pages into usable, compilable C source code.

Overview

ThePBuilder utility uses several input files — in particular, Web content — to generate source files to be used and linked with the `rhphttp` Advanced Web Server (AWS).

The utility allows users to use an HTML file as an input source file. Users can maintain and/or update an HTML page, rerun the PBuilder utility, and recompile the application program to generate updated images. Working in this way, users can directly edit the Web page and debug edits with a standard Web browser, rather than update source code generated from a tool.

The NET+OS API set is shipped with two Web server libraries:

- **Original HTTP server.** Uses the HTML-to-C utility to generate C source code and works well as a basic server.
- **Advanced Web server.** Provides HTTP 1.1 compatibility, file upload capability (based on RFC 1867), file system stub routines, external CGI, use of magic cookies, and Web content compression.

In addition, comment tags (discussed later in this chapter and in the Advanced Web Server Toolkit documentation for the PBuilder utility) lets you add dynamic content such as radio buttons and text boxes into any style of HTML.

The PBuilder utility

The PBuilder utility converts HTML Web pages into usable, compilable C source code. The HTML pages are stored as linked lists of smaller data structures required by AWS. NetSilicon strongly discourages generating these structures manually. The structures are complex, and their internal structure is beyond the scope of this guide.

The PBuilder utility understands special proprietary annotations called *comment tags*. The comment tags are within HTML comment syntax, so they have no effect on the Web page, and they are absent when the page is served by the AWS. However, comment tags allow the user to generate and flesh out hooks (function stubs) with the present dynamic content inserted.

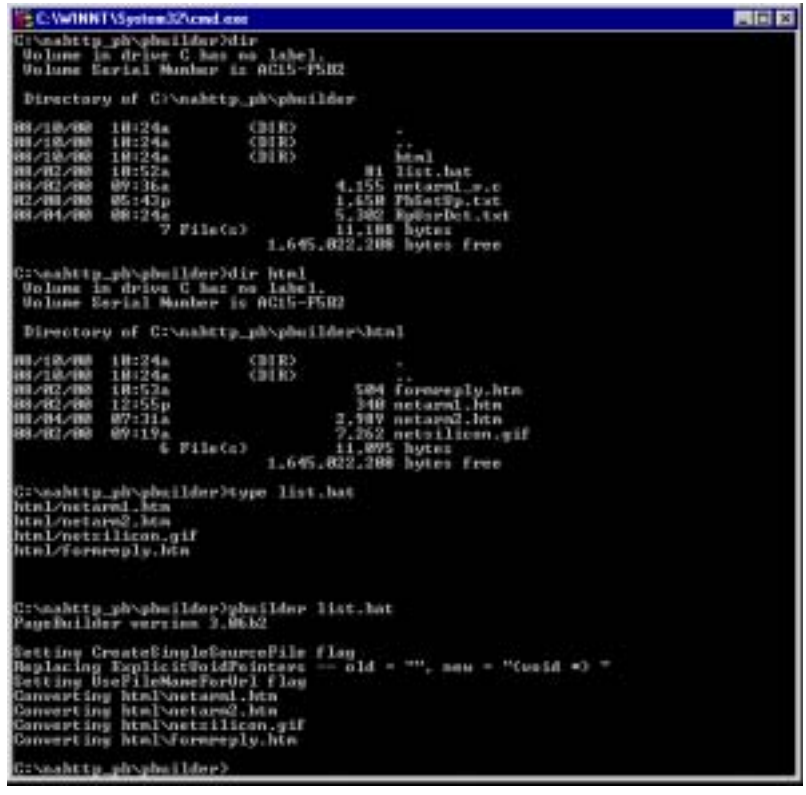
About the Advanced Web Server Toolkit

The Advanced Web Server Toolkit documentation, included on the NET+OS CD, describes how HTML Web content can be annotated with comment tags to help users pass dynamic content through the server. Examples are also provided.

A portion of the guide describes the internal workings of the AWS. These structures and routines are considered private and can be changed at any time. A section also is included that describes the PBuilder utility (PageBuilder Compiler) and how the phrase dictionary is used for Web content compression.

Running the PBuilder utility

Run the PBuilder utility from either PBuilder Helper or a DOS prompt, using the command `pbuilder list.bat`, as shown next:



Explanation of this PBuilder execution

This PBuilder Helper window shows a directory listing followed by a PBuilder execution and the contents of `list.bat`. The file `list.bat` contains all the Web pages used for this application (`nahttp_pb`).

The Web page file (that is, `list.bat`) must have an extension of either `.bat` or `.txt`.

The Web pages within the files are located in the `\html` directory. The `list.bat` file, however, requires the Web pages to be listed with a forward slash; for example, `html/netarm1.htm`.

Two additional files are required to run the PBuilder utility:

- **PbSetUp.txt.** Used to configure the PBuilder utility and should *not* be changed. This file should be copied from the `nahttp_pb` application directory, or will be copied if you are using an Application wizard tool.
- **RpUsrDct.txt.** Contains definitions for Web content compression and is used to generate the `RpUsrDct.c` and `RpUsrDct.h` files. The `RpUsrDct.txt` file can be updated to include common phrases used in the application's Web pages.

The output of this PBuilder execution — `netarm1.c` and `netarm1_v.c` — is located in the `\html` directory and is the source code representation of the Web pages:

- The `netarm1.c` file contains the linked list structures. *Never update or modify this file.*
- The `html\netarm1_v.c` file contains the stubs used for dynamic content. This file was copied to the working directory (`.\`) and fleshed out for this application.

It is good practice to move the `_v.c` files to a different directory. Otherwise, when you run the PBuilder utility again, the fleshed-out version of the file will be overwritten.

This PBuilder execution also produces the file `RpPages.c`, which contains the structure (`gRpMasterObjectList[]`) that contains all the application Web pages.

These files must be compiled and linked for this application:

- pbuilder\html\netarm1.c
- pbuilder\netarm1_v.c
- pbuilder\RpPages.c
- pbuilder\RpUsrDct.c

Because pbuilder\RpUsrDct.h is required, the path \pbuilder\ must be added to the build's include path.

Linking the application with the PBuilder output files

To build an application, you must include the AWS library in the final link of the application. Three additional files must be compiled and included in the build:

- security.c
- file.c
- cgi.c

These files are in the appropriate application directory. You can either leave the files as they are or update them based on Web application requirements. (In the application directory on the NET+OS v5.0 CD, see the sample applications — nahttp_pb or naficgi — for examples of overwriting the files.)

Security.C file

The security.c file can be used to add up to eight security realms. These realms can then be used to password-protect Web pages. For more information, see the nahttp_pb sample application, the *NET+OS Application Software Reference Guide*, or the Advanced Web Server Toolkit documentation for the PBuilder utility.

CGI.C and file.C files

The cgi.c and file.c files are used to handle external CGI and to add or simulate a file system. The file system method was used for uploading and retrieving the file used in the naficgi sample application. For more information about using external CGI, see the naficgi sample application, the *NET+OS*

Application Software Reference Guide, or the Advanced Web Server Toolkit documentation for the PBuilder utility.

Comment tags

The most important aspect of the PBuilder utility is the insertion of comment tags into the HTML Web pages. You can use comment tags to link dynamic data fields with the Web page to specific application variables or functions.

Comment tags are described in detail in the Advanced Web Server Toolkit documentation for the PBuilder utility. NetSilicon strongly recommends that you carefully review the `nahttp_pb` application and read the comment tag section in the PBuilder documentation.

Each comment tag begins with `<!-- RpFormInput... -->` and ends with `<!-- RpEnd -->`.

The Web content within a comment tag (that is, the HTML between `<!-- RpFormInput.... -->` and `<!-- RpEnd -->`) is not used, nor is it required. NetSilicon recommends that you include HTML, however, to assist when you create Web pages.

Creating Web pages

The Management integration with the Advanced Web Server (MAW) API integrates the Advanced Web Server and the Management API. Developers use the MAW API to construct Web pages that access management variables.

The Advanced Web Server (AWS) has a built in way of supporting a custom interface to system variables.

This interface has been adapted to access variables through the management API. This allows developers to use the standard AWS mechanism for embedding dynamic data into Web pages.

This program demonstrates how to create Web pages that display and change management variables.

AWS custom variables

AWS allows developers to create Web pages that can display the current value of variables and prompt users for new values. To do this, developers insert comments in their HTML pages that have special tags that AWS recognizes. These tags identify variables to AWS and tell it how to access them. For example, these HTML comment contains tags tell AWS to display the variable named Username:

```
<!-- RpNamedDisplayText Name=Username RpTextType=ASCII
RpGetType=Function
RpGetPtr=GetUsername -->
<!-- RpEnd -->
```

The HTML comment starts with the keyword RpNamedDisplayText, which identifies the HTML comment as an AWS command to insert the current value of a variable into a Web page. The tag Name=Username tells AWS the variable is named Username, the RpTextType=ASCII tag tells AWS that the variable is an ASCII string, the RpGetType=Function tag tells AWS that a function has been supplied to read the variable, and the RpGetPtr=GetUsername tag tells AWS that the function is named GetUsername. When AWS encounters this tag, it calls the `function GetUsername()`, which returns an ASCII string that AWS inserts into the Web page. The use of AWS tags is explained in detail in the Advanced Web Server Toolkit documentation for the PBuilder utility.

AWS normally accesses variables directly through either pointers or functions that the developer writes. However, AWS also has a built-in mechanism to support customized access to variables. Comment tags that use the custom interface for accessing variables are similar, except that the RpGetType and RpSetType tags must be set to Custom, and the RpGetPtr and RpSetPtr tags are no longer needed. Setting the type tag to Custom tells AWS to call a customizable routine to get the value of the variable.

AWS has been integrated with the management API by modifying AWS's customizable routines to access management variables. So, for example, if the variable Username was registered with the management API, the AWS comment tag to display its value would be:

```
<!-- RpNamedDisplayText Name=Username RpTextType=ASCII
RpGetType=Custom -->
<!-- RpEnd -->
```

Data types

AWS data types are mapped to management API data types in this manner:

AWS type	Management type
ASCII	MAN_CHAR
ASCIIExtended	MAN_CHAR
ASCIIFixed	MAN_CHAR
HEX	WORD8
HEXColonForm	WORD8
DotForm	WORD8
Signed8	INT8
Signed16	INT16
Signed32	INT32
Unsigned8	WORD8
Unsigned16	WORD16
Unsigned32	WORD32

Displaying variables

Developers use AWS RpNamedDisplayText comment tags to display the values of management variables in Web pages. The comment tags take this form:

```
<!-- RpNamedDisplayText Name=name RpTextType=type RpGetType=Custom-->
<!-- RpEnd -->
```

where

- *name* must be replaced with the name of the management variable to display.
- *type* must be replaced with the AWS type of the variable.

For example, suppose monthString is a character string, yearInt32 is a 32-bit integer, and dayWord8 is an 8-bit word, and that all the variables have been registered with the management API. The HTML code to display them would be:

```
The date is
```

```
<!-- RpNamedDisplayText Name=monthString RpTextType=ASCII
RpGetType=Custom -->
<!-- RpEnd -->
<!-- RpNamedDisplayText Name=dayWord8 RpTextType=Unsigned8
RpGetType=Custom -->
<!-- RpEnd -->
,
<!-- RpNamedDisplayText Name=yearInt32 RpTextType=Signed32
RpGetType=Custom -->
<!-- RpEnd -->
```

Changing variables

Developers use HTML forms to prompt users for input. AWS comment tags are embedded in the HTML form commands to tell AWS how to transfer the user's input into application variables. The RpFormInput tag can be used to prompt users for a numeric value or a string. The format of this tag is:

```
<!-- RpFormInput TYPE=promptType RpTextType=dataType NAME=name
RpGetType=Custom RpSetType=Custom MaxLength=length Size=size -->
          html code
<!-- RpEnd -->
```

where:

- *promptType* is replaced with the type of prompt for this input field (text, password, hidden, checkbox, or radio button)
- *dataType* is replaced with the AWS data type for the variable
- *name* is replaced with the name the variable was registered under
- *length* is replaced with the maximum length for the variable
- *size* is replaced with the size of the input field

For example, this HTML code prompts the user to enter a value for maxTemperature, which is a 16-bit integer. The prompt is a 15-character wide text field.

```
<!-- RpFormInput TYPE=text RpTextType=Signed16 NAME=maxTemperature
RpGetType=Custom RpSetType=Custom MaxLength=15 Size=15 -->
<!-- RpEnd -->
```

Developers can use the RpFormTextAreaBuf comment tag to prompt the user for a string value with a multi-line text box. This takes the following form.

```
<!-- RpFormTextAreaBuf NAME=name RpGetType=Custom RpSetType=Custom
ROWS=height
COLS=width -->
<!-- RpEnd -->
```

where

- name is replaced with the variable's name
- height is replaced with the height of the text box
- width is replaced with the width of the text box

For example, this HTML code prompts the user with a 4 by 50 text box to enter a new value for the string postalAddress:

```
<!-- RpFormTextAreaBuf NAME=postalAddress RpGetType=Custom
RpSetType=Custom ROWS=4
COLS=50 -->
<!-- RpEnd -->
```

Developers can also use a select list to prompt for a numeric value to be written into an 8-bit word. The RpFormSingleSelect and RpSingleSelectOption tags are used for this. The RpFormSingleSelect tag sets up a select list. The RpSingleSelectOption tag sets up individual items in the select list. The RpFormSingleSelect tag has this form:

```
<!-- RpFormSingleSelect NAME=name RpGetType=Custom RpSetType=Custom
Size=size -->
option list
<!-- RpEnd -->
```

where

- *name* is replaced with the variable's name
- *size* is replaced with the number of visible lines in the select list
- *option* list is replaced with a list of RpSingleSelectOption tags

The RpSingleSelectOption tag has this form:

```
<!-- RpSingleSelectOption value="text label"
RpItemNumber=numericValue -->
<!-- RpEnd -->
```

where text label is a label for this option, and numericValue is the corresponding numeric value to be assigned to the variable if the user selects this option.

This example sets up a select list that prompts the user to choose a day of the week. The variable dayOfTheWeek is set to a value between 0 and 6, depending upon which day the user chooses.

```
<!-- RpFormSingleSelect NAME=dayOfTheWeek RpGetType=Custom RpSetType=Custom Size=7
-->
<!-- RpSingleSelectOption value="Sunday" RpItemNumber=0 -->
<!-- RpEnd -->
<!-- RpSingleSelectOption value="Monday" RpItemNumber=1 -->
<!-- RpEnd -->
<!-- RpSingleSelectOption value="Tuesday" RpItemNumber=2 -->
<!-- RpEnd -->
<!-- RpSingleSelectOption value="Wednesday" RpItemNumber=3 -->
<!-- RpEnd -->
<!-- RpSingleSelectOption value="Thursday" RpItemNumber=4 -->
<!-- RpEnd -->
<!-- RpSingleSelectOption value="Friday" RpItemNumber=5 -->
<!-- RpEnd -->
<!-- RpSingleSelectOption value="Saturday" RpItemNumber=6 -->
<!-- RpEnd -->
<!-- RpEnd -->
```

Security

The MAW module supports the NET+OS security API and the security features built into AWS. AWS allows developers to associate a username and password with a group Web pages. The combination of the username, password, and list of Web pages is called a *realm*. The AWS forces users to supply the correct username and password whenever they access any page in the realm. Up to eight different realms can be created.

Exceptional cases

It may be necessary to write special purpose code to access management variables. In these cases, developers can specify the AWS Function type in the comment tags, and then supply functions to perform the access. In this example, the functions appGetDate() and appSetDate() are defined to access a management variable.

```
<!-- RpFormInput TYPE=text NAME=dateString RpGetType=Function
RpGetPtr=appGetDate
RpSetType=Function RpSetPtr=appSetDate MaxLength="31" Size="31" -->
<!-- RpEnd -->
```

Controlling the MAW module

The MAW module can be configured to control:

- What timeout is used to access management variables
- What array subscripts are used when accessing management variables that are arrays
- How error conditions are handled

Default configuration

Parameter	Default action
Semaphore timeout	Wait forever for semaphores to unlock.
Array subscripts	If the variable is a one dimensional character array, read or write the entire variable.
Error handling	Halt system on errors.

Setting the semaphore timeout

Management variables can be protected by one or more semaphores. When the MAW module accesses a management variable, it specifies the maximum amount of time it will wait for the semaphores to unlock. By default, the timeout is infinity.

Applications change the timeout value with the function `mawSetAccessTimeout()`,defined as:

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

The parameter timeout specifies the new timeout value.

Alternatively, applications can specify different timeouts for each variable. To do this, use the `mawInstallTimeoutFunction()` function to register a function that is

passed the name of each variable being accessed, and returns the appropriate timeout. This function is defined as:

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

The appFunction is a pointer to a function supplied by the application that will control the timeouts used for each variable.

`mawTimeoutFn Type`

The type mawTimeoutFn is defined as:

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

The parameter varName specifies the name of the variable being accessed, and the function returns an appropriate timeout value.

Array subscripts

If Web pages access management variables that are arrays, the application must register a function to specify the appropriate subscripts to use when the arrays are accessed. This is done by calling the function `mawInstallSubscriptsFunction()`, which is defined in this way:

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

The parameter appFunction is a pointer to the application-supplied function that determines the subscripts that will be used.

mawSubscriptsFn Type

The `mawSubscriptsFn` type is defined in this way:

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

This table shows the parameters and their descriptions:

Parameter	Description
varName	Name of the variable being accessed
indices	A pointer to an array of integers that are the current loop indices being used by the HTTP server
dimensions	A pointer to an array of integers that specify the dimensions of the management variable

Parameter	Description
numberDimensions	The number of dimensions the management variable has
htmlType	The data type the HTTP server is expecting

The function must return a pointer to an integer array that contains the subscripts of the array element to be accessed. If the variable is an array of character, the function may return NULL to indicate that the entire array is to be read or written.

Error handling

Applications can use the `mawInstallErrorHandler()` function to install an error handler. This function is defined as:

```
void mawInstallErrorHandler (mawErrorFn appFunction);
```

The parameter *appFunction* is a pointer to the application's error handler.

mawErrorFn Type

The type `mawErrorFn` is defined in this way:

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

This table lists the parameters and their definitions:

Parameter	Description
VarName	Name of the variable being accessed
HtmlType	Data type expected by the HTTP server
Error	Identifies the error condition

The function should either halt the system or return a value that the HTTP server can use.

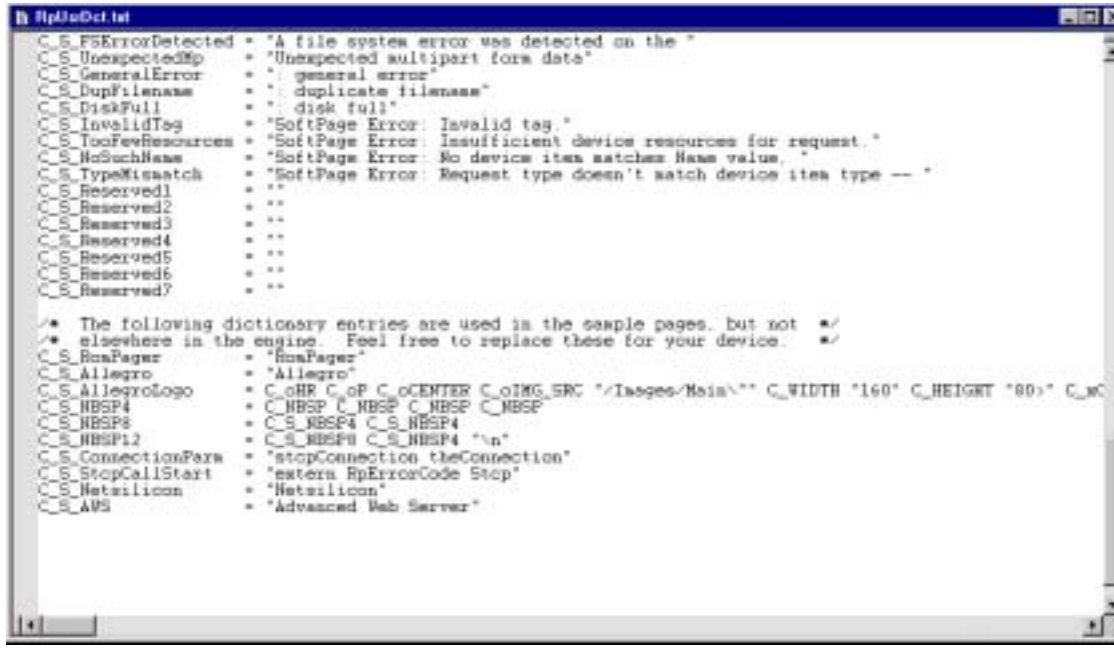
Building the application

The MAW module is built into a special version of the AWS library. The application should be linked against this version of the AWS library.

Phrase dictionaries and compression

The AWS uses a phrase dictionary technique to provide compression for static ASCII text strings with the HTML Web content. The PBuilder utility uses the `RpUsrDct.txt` file as an input and builds its data structures to point to common phrases in the dictionary instead of repeating strings.

This figure shows the content of the `RpUsrDct.txt` file for the `nahttp_pb` application:



Common phrases within all the application Web pages should be added (for example, a company name that is used several times). In the sample file, note this definition:

```
C_S_AWS = "Advanced Web Server"
```

This definition is used several times within the application Web pages. Search the file `\pbuilder\html\netarm1.c; the string C_S_AWS` is used consistently throughout the file. This is an example of compression.

Maintaining and modifying Web content

After application source files have been generated, the best way to maintain and update Web content is through the HTML pages. NetSilicon recommends that you maintain these files and include them in source control.

If a Web page requires a change or an additional new page, you should either update the HTML, add a new page to the `list.bat` file, or do both. New phrases can be added to the dictionary at any time.

For the changes to take effect, rerun the PBuilder utility. The application or image is automatically rebuilt.

Sample applications

Two sample applications are included in the application directory:

- **nahttp_pb.** This application shows several examples of using comment tags, overwrites the `security.c` file to use a password-protected page, and shows an example of the phrase dictionary.
- **naficgi.** This application shows how a file can be uploaded and served, and overwrites the `cgi.c` and `file.c` files to external CGI.

Troubleshooting

CHAPTER 8

This chapter describes how to diagnose some problems you may encounter when you are working with NET+OS.

Crash for debug

Use caution when you debug new applications or modify board support packages, because errors such as Pre-fetch Abort, Data Abort, and Undefined Instruction c that cause exceptional processor conditions can occur. It is essential that you can recover from these conditions and extract postmortem data. The most important data item extracted is the statement that caused the crash.

The `crash_for_debug` routine is used in several places within the board support package after *exceptional conditions* have occurred. Exceptional conditions include:

- Routine loops continuously
- Blinking the red LED
- Excessive wait times

When you run the application from the debugger, you can extract crucial postmortem information from the `crash_for_debug` routine in any of several ways. For example, in the routine, you can break into the debugger and examine memory. Although this is helpful, you still need to discover the point of failure.

▶ **To find the statement after the point of failure:**

1. Break into the debugger.
2. Step out of the function you are in until it returns to `crash_for_debug`, unless control is already in `crash_for_debug`.
3. In the debugger, change the value of `crash_exit` to 1.
4. Step out of the `crash_for_debug` routine.
5. Continue stepping to exit from the exception handler in `init.s`.

When you exit from the exception handler, you will be at the next instruction, which is the statement after the one that caused the crash. After you locate the point of failure, you should be able to discover what caused the problem.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

W


NetSilicon
A Digi International® Company