NET+Works with Green Hills Programmer's Guide

Making DEVICE NETWORKING easy™

NET+Works with Green Hills Programmer's Guide

Operating system/version: 6.3
Part number/version: 90000723_B
Release date: March 2005
www.digi.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 6.3 with Green Hills Tools 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.

Software release

This guide supports NET+OS 6.3. By default, this software is installed in the `C:/netos63_ghs/` directory.

Who should read this guide

This guide is for software engineers and others who use NET+Works 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 name→ menu selection name**	Menu commands. The first word is the menu name; the words that follow are menu selections.
`monospaced type`	File names, pathnames, and code examples.

What's in this guide

This table shows where you can find information this guide:

To read about	See
The NET+Works components	Chapter 1, "NET+Works Introduction"
Converting HTML and related Web page files to C code	Chapter 2, "Using the HTML-to-C Compiler"
Using the MIBMAN utility to implement management information bases (MIBs)	Chapter 3, "Using the MIBMAN Utility"
Using the PBuilder utility to convert HTML Web pages into compilable C source code	Chapter 4, "Using the Advanced Web Server PBuilder Utility"
Diagnosing problems; reserializing a development board	Chapter 5, "Troubleshooting"

Related documentation

- *NET+Works Quick Installation Guide* describes how to install the hardware.
- *Green Hills MULTI 2000 IDE Licensing Information* describes how to request a license key.
- *NET+Works with Green Hills Tutorial* provides a brief, hands-on exercise.

- *NET+Works with Green Hills BSP Porting Guide* describes how to port the board support package (BSP) to a new hardware application.
- The NET+Works online help describes the application program interfaces (APIs) that are provided with NET+OS. By default, the online help is located in:
  `C:\netos63_ghs\Documentation`

For information about third-party products and other components, review the documentation CD-ROM that came with your development kit.

For information about the processor you are using, see your NET+Works hardware documentation.

Documentation updates

Digi occasionally provides documentation updates on the Web site. Be aware that if you see differences between the documentation you received in your NET+Works package and the documentation on the Web site, the Web site content is the latest information.

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 here:

- United States telephone: 1 877 912-3444
- International telephone: 1 952 912-3444
- email: digi.info@digi.com
- Web site: http://digi.com

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 are being implemented into product designs.

NET+Works is designed as an alternative to a PC for products that must be connected to Ethernet for Transmission Control Protocol (TCP) access and for Internet/Intranet access.

The NET+Works package includes:

- A NET+Works processor
- A development board and board support package
- If you are using a hardware debugger, either a MAJIC probe or a Raven debugger
- The networking firmware
- The object code with application program interfaces (APIs)
- Development tools
- Sample code
- Documentation

For information about the NET+ARM devices, see the hardware documentation.

System components

This section describes the components that make up the NET+Works software.

NET+Works runtime software

NET+Works software provides the building blocks to help you create your custom applications. You create your application with calls to APIs for:

- The board support package (BSP)
- ThreadX RTOS kernel
- Basic Internet protocols
- Higher-level protocols and services

Board support package

The NET+Works BSP is a collection of ARM object code and C source-code drivers and bootloader. The BSP initializes hardware and software, and it provides power-on self test (POST).

The BSP includes a set of APIs that you use to incorporate device peripheral functionality into your application. In addition, the BSP provides the drivers for your NET+ARM processor or Digi Connect module development board, including Ethernet, Serial, SPI, Flash, USB host, USB device, LCD, PCI/CardBus, and others.

ThreadX RTOS kernel

The RTOS is based on a high-speed picokernel architecture. ThreadX helps you manage complex event synchronization and memory using threads, queues, application timers, semaphores, and event flags.

HTML-to-C compiler

The toolkit provides a compiler that translates your HyperText Markup-Language (HTML) code into C code. You can then compile the C code as part of the project, which allows you to create the pages your menu system needs to perform the tasks necessary for your device. Sample code also is provided.

Advanced Web Server (AWS) PBuilder utility

Using the PBuilder utility, you create or maintain Web pages and recomplile the application program to generate updated images. The PBuilder provides support for HTML, multiple Web object sources, object compression, and advanced security.

Address Configuration Executive (ACE)

NET+Works provides services such as the ACE, which lets you acquire IP parameters at startup from multiple prioritized sources, including BOOTP, RARP, Auto IP, and others.

System requirements

To run NET+Works with Green Hills, your system must meet these requirements:

- A minimum of a 700 MHz Pentium processor and 256 MB of RAM. A 1.4 GHz Pentium processor with 512 MB of RAM is recommended.
- A 256-color display, at a minimum resolution of 1024x768. A minimum resolution of 1280x1024 is recommended.
- An Ethernet connection.
- 1000 MB of free disk space per installation. If an older version of MULTI is already installed on your machine, install this version in a different directory. If you are installing to a drive other than your primary Windows drive, you also must have at least 10 MB available on the primary drive for temporary files and DLLs.
- A CD-ROM drive.

You need this software installed on your system:

- Internet Explorer 3.01 or later (to view online help)
- Microsoft Windows 2000 or XP

Using the HTML-to-C Compiler

CHAPTER 2

This chapter describes the HTML-to-C compiler, which you use to convert HTML and related Web page files to C code.

Overview

This chapter describes the HTML-to-C Compiler, which is one of two Web authoring tools supplied with NET+Works. The HTML-to-C Compiler allows you to develop simple web pages.

Another web authoring tool, the Advance Web Server, allows you to create more sophisticated Web content and provides a way to translate web pages into foreign languages. For more information about the Advanced Web Server, see Chapter 4.

The HTML-to-C compiler converts HyperText Mark-up Language (HTML) and related Web page files to standard C code so you can compile and link the Web pages for an application.

The HTML-to-C compiler provides an easy way to integrate Web pages and content into the Web server. Components converted to C code are easily integrated into NET+Works software applications.

The next sections provide background information about HTML and describe how to use the HTML-to-C compiler.

Web content

In response to HyperText Transport Protocol (HTTP) requests from Web browsers, devices send *Web content*, which consists of

- HTML pages
- Images (such as `.gif` and `.jpeg` files)
- Java applets
- Audio files

To incorporate Web content into an embedded device, you first build an HTTP server into the device by using the application program interfaces (APIs) provided with NET+Works (described in the online help). 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 systems have storage disks with large file systems. Incorporating Web content is 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 develop pages by adding HTML markup tags to the text content, using either a text editor or Web authoring tool. You can add the same pages to an embedded device by writing application code to physically return an HTML page. The page is stored in a large character buffer in the device and returned through a network API such as sockets.

As tools for generating HTML pages become more advanced, webmasters do not generate HTML pages by hand. Web authoring tools are more efficient and reduce the amount of typing needed in markup tags.

Static and dynamic content and forms processing

Embedded devices must be able to incorporate *static* content, *dynamic* content, and *forms processing* into the embedded HTTP server source code.

The HTML-to-C compiler 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. Nothing needs to be added after the page is converted. Static pages are a small part of the content provided by a Web server.
- *Dynamic* content, which changes over time, is necessary for status reporting. This type of content has proprietary non-HTML markup tags inserted with an HTML editor into the HTML source code. The HTML-to-C compiler 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 routines so that the appropriate dynamic content is returned when the routines are called.
- *Forms processing*, which accepts user input and acts on it, also is necessary for making 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.

The HTML-to-C compiler also recognizes HTML form tags and adds the shell routines to be called when forms data is sent back to the embedded Web server.

The embedded Web server typically has an 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.

Dynamic content example

Sometimes you want a Web page to look different every time it is accessed. For example, you may want to write a page that provides weather or traffic reports. Dynamic content is useful in such a case. All you have to do is to embed the `_NZZA_` tag into your HTML file. The HTML-to-C compiler understands this tag and generates an empty routine at the end of the generated C file.

This example shows a portion of a Web page that reports the temperature at Logan Airport:

```
<h1> The temperature at Logan is _NZZA_get_temp </h1>
```

This is translated into statements:

```
HSSend (handle, “<h1> The temperature at Logan is”);
na_get_temp(handle);
HSSend (handle, “</h1>”);
```

Note that `_NZZA_get_temp` is being converted into the `na_get_temp` function call, 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_airport_temp();
```

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

Input form example

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 compiler 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 character s. 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`, as shown next:

```
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   */
}
```

Preparing to use the HTML-to-C compiler

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

To incorporate Web content into an embedded device, you first build an HTTP server into the device using the APIs provided with NET+Works.

▶ **To build an HTTP server into a device:**

1. Create Web pages by using an HTML editor.
2. Use the HTML-to-C compiler to convert the Web pages into C code.
3. If the Web pages contain dynamic content or forms, add your own code to the generated empty C routines.
4. Put all your C source files into your application's `project.gpj` file.

How the HTML-to-C compiler works

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

- **Text files.** Any file with an extension of `.html`, `.htm`, or `.txt`. A text file causes the compiler to generate a .c file with a similar file name.
- **Binary files.** All binary files are converted and stored in the `bindata.c` file. All the URL information, including security, is stored in the `url.c` file.
  The file names `bindata.c` and `url.c` are default names you can change.

For example, assume 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 url.c file:

```
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 */
};
```

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

Binary data, however, is treated differently because it does not need to be customized. 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 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 Send_function_1 in the file _~John_home.c function:

```
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”);
```

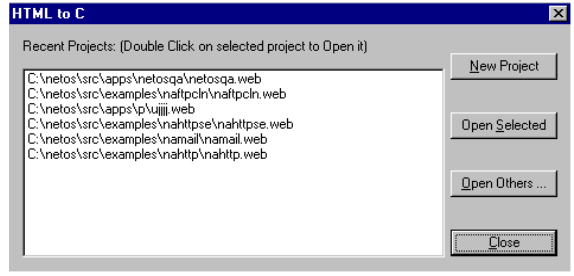
```
HSSend (handle, “<meta name=\”GENERATOR\
    “content=\”Microsoft FrontPage Express 2.0\“>\n”);
HSSend (handle, “<title>John’s Home Page</title>\n”);
HSSend (handle, “</head>\n”);
HSSend (handle, “\n”);
HSSend (handle, “<body bgcolor=\”#FFFFFF\“>\n”);
HSSend (handle, “\n”);
HSSend (handle, “<h3>Welcome to John’s Home </h3>\n”);
HSSend (handle, “\n”);
    HSSend (handle, “</body>\n”);
    HSSend (handle, “</html>\n”);
    }
```

Using the HTML-to-C compiler

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

▶ **To start the HTML-to-C compiler:**

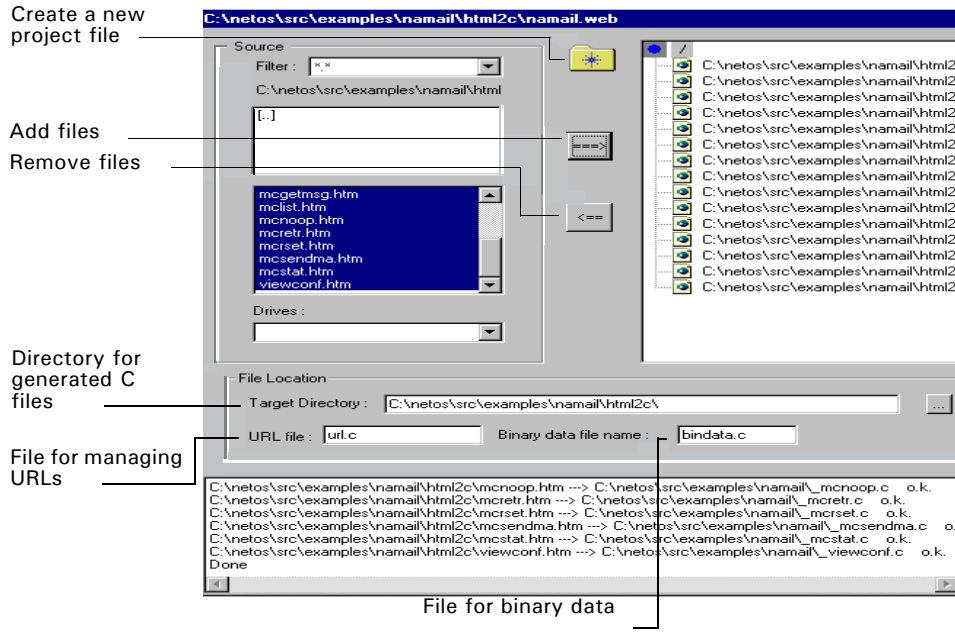
1 Select **Start → Programs → netos63_ghs → HTML-to-C Compiler.**
The **HTML to C** dialog box opens:



2 Do one of these steps:

- To open a project that appears in the list, select the project from the list and click **Open Selected**.
- To open a project that does not appear in the list, click **Open Others**, then locate and select the project.
- To create a new project, click **New Project**, then enter a name and select a location for the project.

This dialog box opens:



From this dialog box, you can:

- Create a new project.
- Remove obsolete data.
- Specify locations for generated C files and names for binary data and URL files.
- Generate C source files.
- Open and edit the `url.c` file.
- Set or change the project's home page.

Creating a new project

▶ **To create a new project:**

1. From the main dialog box, click the folder icon in the **Source** section.
2. Enter a name and select a location for the new project.

Removing obsolete data

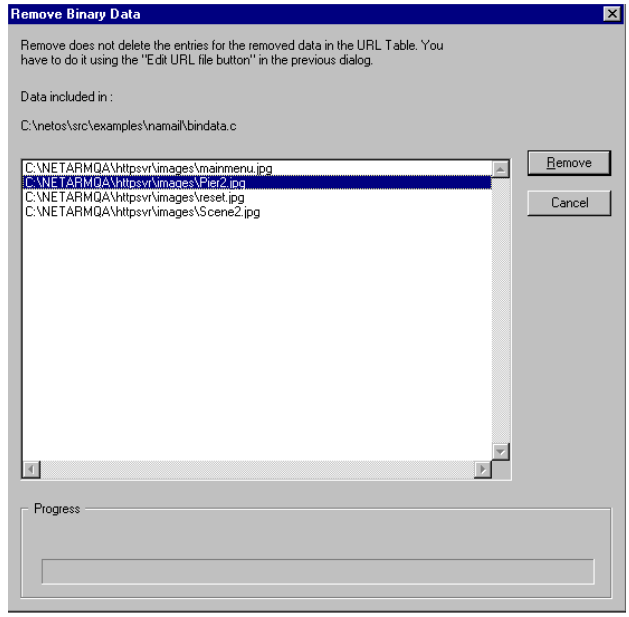
Removing obsolete data from a project file reduces compilation time. Use caution when you remove files; you may need the data for troubleshooting.

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

▶ **To remove obsolete data from a file:**

1 From the main dialog box, click **Remove Bin data**.

The **Remove Binary Data** dialog box opens, and 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**.

Adding or removing source files

You can add or remove source files at any time.

▶ **To add a source file:**

1. In the **Source** section of the main dialog box, locate and select the file to add to your project.
2. Click the right arrow.

▶ **To remove a source file:**

1. In the **Source** section of the main dialog box, locate and select the file to remove from your project.
2. Click the left arrow.

Specifying the location of files

From the main dialog box, you can specify the location for the generated C files, and the names of the URL and binary data files, as described here:

- To specify the directory in which to put the generated C files, enter the path name in the **Target Directory** input box.
- To specify a file name for the URL file, enter the name in the **URL file** input box. The default name is `url.c`.
- To specify a file name for the binary data file, enter the name in the **Binary data file name** input box. The default name is `url.c`.

  If your project does not have any binary files, you do not need to include this file in your `project.jpg` file.

Generating C source files

From the main dialog box, you can generate C source files in two ways:

- To generate *only* the files that have changed since the last build, click **Build**.
- To generate all files, click **Build All**.

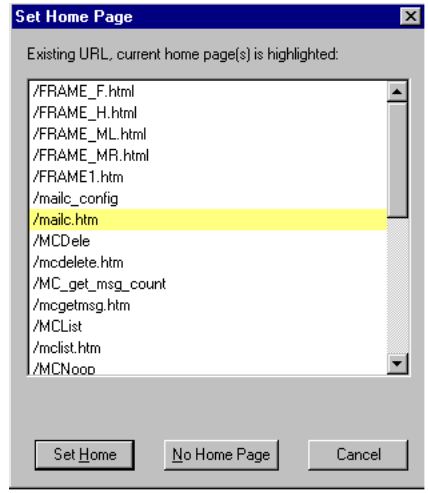
Setting or changing a project's home page

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

▶ **To set or change the home page of a project:**

1 From the main dialog box, click **Home Page**.

The **Set Home Page** dialog box opens:



2 Select the URL of the page you want to use as home page and click **Set Home**.

If you do not want to specify a home page, click **No Home Page**.

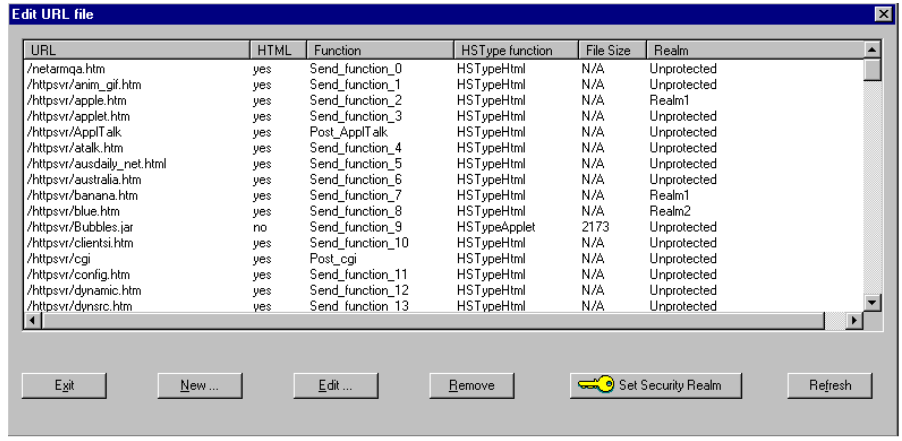
Editing URL files

The `url.c` file contains the URL table, from which you can:

- Add or delete entries.
- Edit entries.
- Set or change security realms.

Opening the url.c file

To open the `url.c` file, from the main dialog box, click **Edit URL file**. The **Edit URL file** dialog box opens:



Adding and deleting URLs

▶ **To add a new URL:**

1. From the **Edit URL file** dialog box, click **New**.

   The **Edit URL** dialog box opens.

2. Provide the necessary information, and click **OK**.

▶ **To delete a URL:**

1. From the **Edit URL file** dialog box, select the URL to delete.
2. Click **Remove**.

   Be aware that the deletion happens immediately; you do not get a prompt to confirm the deletion.

Editing a URL

▶ **To edit a URL:**

1. From the **Edit URL file** dialog box, select the URL you want to edit.
2. Click **Edit**.

   The **Edit URL** dialog box opens.

3. Make the changes you want, and click **OK**.

Setting the user and password of a URL

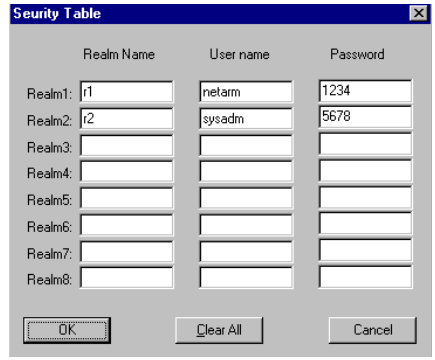
The HTTP server supports the basic access authentication scheme in the HTTP protocol. The combination of the username, password, and list of Web pages is a *realm*. You can create up to eight realms.

You must set up a security table before you start the HTTP server.

▶ **To set up or change realm information:**

1 From the **Edit URL file** dialog box, click **Set Security Realms**.

The **Security Table** dialog box opens:



2 For each realm, type the realm name, user name, and password, and then click **OK**.

Using the MIBMAN Utility

CHAPTER 3

This chapter describes the MIBMAN utility, which you use to implement management information bases (MIBs).

Overview

MIBMAN is a utility that translates Simple Network Management Protocol (SNMP) Management Information Bases (MIBs) into C code that contains:

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

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

This chapter describes how to convert an SNMP MIB into C code and how to write action routines.

Terms and concepts

The next sections provide some terminology you need to become familiar with The next sections provide some terminology to be familiar with as you use MIBMAN.

SNMP

SNMP, which defines a system for managing network devices, is implemented in multiple 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. You implement the subroutines, which are called *action routines*.

A MIB defines the view of a managed device that the device's agent presents to an SNMP console.

The MIB defines a set of objects the console can read and write. These objects can either correspond to variables that exist on the device or be synthesized by the agent:

- MIB objects that represent variables that exist are called *real objects.*
- MIB objects that do not exist are called *virtual objects*.

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 agent can determine the health of the system on-the-fly by examining the state of several system variables and processes.

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.

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

MIBMAN cannot always use generic action routines. 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 such a 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

You can arrange MIB objects 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
- Virtual objects that 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
- 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; you implement the table's indexing scheme. For more information, see the comments in the MIB and the RFCs that describe the MIB.

Tables can have relationships with other tables; for example:

- One table can expand another table by defining additional columnar objects. In this case, the two tables have a one-to-one relationship.
- One 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.
- One table can be a sub-table of another. Implementing this relationship requires complex indexing schemes.

These relationships can cause the fate of rows in one table to affect rows in another table. For example, assume one table expands another. When a row in one table is deleted, the corresponding row in the other table also must be deleted. These types of relationships, which you implement, are not described by the MIB, but in comments in either the MIB or the RFCs that describe the MIB.

MIBMAN can create management variable declarations and action routine templates for MIB tables that represent real tables in memory. You must initialize the tables and update the data in them as necessary. You also must 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 that an agent sends to a console when certain events occur. Although the contents of traps are defined in 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.

You can use MIBMAN to generate a set of data structures that applications use to generate and send traps.

Action routines

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

Implementing a MIB: an example

`namib`, a sample application on the NET+Works installation CD, demonstrates how to implement a MIB. The sample MIB allows you to read and change the state of the LED on the board from a MIB browser. `namib` also demonstrates how to implement a MIB with tables.

Converting an SNMP MIB into C code

To convert an SNMP MIB into C code, you use this general procedure:

1. Use a MIB compiler to convert the MIB definition into an intermediate file.

   To compile the MIBs, you can use the SMICng MIB compiler. SMICng, an industry-standard compiler, is shipped with the SNMP agent.

2. Run MIBMAN to convert 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.

3. Do the final integration.

The next sections provide details about the three steps.

Step 1: Using SMICng

Because the developers of SNMP tried to design a format for SNMP MIB definitions that would be readable by both machines and people, MIB definitions are difficult for anyone or anything to read. The SMICng compiler, however, converts a MIB definition into a form that is easier for software to read.

For more information about SMICng, see the SMICng online documentation.

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, you also need to clean up the text files that contain the definitions for these MIBs.

2. Create an include file for SMICng that lists all the MIBs used by the one that is being processed.

   Include the MIB itself as the final file.

3. Run SMICng with this command line:

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

   where:

   - *`yourMib.inc`* is the name of the include file you created in step 2 of this procedure.
   - *`yourMib.out`* is the name of the intermediate file the compiler creates.

Step 2: Using MIBMAN

The MIBMAN utility operates on the intermediate files that SMICng creates. MIBMAN accepts up to four arguments, as shown here:

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

where:

- *`list-file`* is the file that contains a list of intermediate files generated by SMICng.
- *`configuration-directory`* is the directory in which configuration files are located.
- *`output-directory`* is the directory in which output files are generated.
- *`var-filename`* is the file that MIBMAN creates to contain a list of the variables defined by the MIB and their OIDs.

The *`list-file`* argument is required; the other three arguments are optional. If you do not specify the optional arguments, MIBMAN uses the current directory and does not generate a list of MIB variables. In addition, 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, assume RFC1213-MIB is 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.OUT
RFC1212-OUT.OUT
RFC1213-OUT.OUT
```

Generated files

MIBMAN generates two C files and a header file (.h) for each MIB it processes that defines variables or traps. Here are the naming conventions:

- By default, the name of the MIB module determines file names.
- MIBMAN converts the MIB module name into legal Windows file names by changing hyphens and periods in the name into underscores and appending the extensions .c and .h.
- The suffix *Action* is appended to the template file name.
- If the module name starts with a digit, an underscore is added to the beginning of the file name.

One C file —the *definition file* — contains the definitions for variables and structures that the management API and the SNMP agent use, 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 you use 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 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. You can change these default settings through a configuration file.

Data types

SNMP data types are represented by similar management variable data types, as shown in this table:

These SNMP data types	Are represented as
Textual conventions, octet strings, and opaque objects	`MAN_OCTET_STRING_TYPE` variables.
`INTEGER` and `Integer32`	`INT32` types.
`Unsigned32`, `Gauge`, `Gauge32`, `Counter`, `Counter32`, and `TimeTicks`	`WORD32` types.
`Counter64`	`WORD64` type.
`OBJECT IDENTIFIER`	Arrays of 129 `WORD32` elements. The length of the OID is encoded in the last element of the array.
`IPaddress` and `NetworkAddress`	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

MIBMAN generates code to register tables with the management API by 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 the next 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 a `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. You can suppress entries for virtual objects.

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

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.

Field	Description
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, this field is set to NULL.
tableInfo	Set to NULL if the object is 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 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, you 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:

- *id* is loaded with the trap's OID.
- *trapNumber* is set to the trap number.
- *trapType* is set to either of these:
  - *TRAP_TYPE* if the trap is an SMI version 1 trap
  - *NOTIFICATION_TYPE* if the trap is an SMI version 2 notification.
- *variableCount* is loaded with the number of variables in the trap.
- *varList* is a pointer to a list of variables.

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

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

where:

- *id* is the ID of the variable (usually its OID).
- `hintCode` is a value passed to some functions in the Fusion agent to identify columnar objects in a row.
- *oid* is the OID of the variable.
- *type* is a constant defined in ASNI1.H that indicates the data type of the variable.

MIBMAN configuration file

You can override some of the default values MIBMAN uses through a configuration file. Here are the naming conventions:

- The MIB that is being processed determines the configuration file's name.
- MIBMAN converts the MIB module name into legal Windows file names by changing hyphens and periods in the module name into underscores.

- The suffix .config is appended.
- If the module name starts with a digit, an underscore is prefixed to the file name.

For example, assume 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 file name RFC1213_MIB.config.

If the configuration file does not exist, the default values 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 must 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. You can override these names.

To set the name of the .c and .h files, use these statements:

- Cfilename *c-file name*
- Hfilename *h-file name*
- Actionfilename *action-file name*

where you replace *c-file name*, *h-file name*, and *action-file name* with your file names.

Overwriting files

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

Controlling comments in the C and header files

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

Option	Description
`Author` *`author`*	Sets the author listed in the comments of the C and header files.
`ModuleName` *`modulename`*	Overrides the module name specified in the MIB.
`Description` *`description`*	Sets description information. You can specify any number of description options.
`Edit EditId=`*`text`*	Lets you keep an edit history. You specify an edit ID and text to go along with it. The edit ID can be up to 14 characters. The date of the change is a useful 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 the example:

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

Controlling management variables

Suppressing management variables

To prevent MIBMAN from generating a management variable declaration for a MIB object, use `DontCreateVariable`.

`DontCreateVariable` *`oid`*

This option is useful if the object 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`*

You can specify prefixes for global variables that MIBMAN creates. Using these options can help avoid collisions with other global variables in the application.

- `MibVariablePrefix` — for variables that are related to MIB items
- `AgentVariablePrefix` — for global variables that are not related to a specific MIB

Setting the management ID

`SetVariableIdentifier` *`oid = identifier`*

To specify the management API's identifier for the variable, use the `SetVariableIdentifier` option. By default, MIBMAN uses the object's OID represented as a character string. You can use this option 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 can use complex indexing systems. By default, MIBMAN creates the variable without an index system, which means rows are read using a simple numeric index. If this is not sufficient, you can use the `SetIndexFunction` and `SetIndexInfo` options to specify a complex indexing system:

- `SetIndexFunction` sets the name of a function that the management API uses to index into the table.
- `SetIndexInfo` 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

To specify that some or all the variables are to be protected by semaphores, use the `SetGlobalSemaphore` and `SetSemaphore` options.

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

- `SetGlobalSemaphore` sets a semaphore that protects 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 a good practice to create custom action routines for real scalar objects when reading or writing them is intended to cause side effects.

To generate templates for these routines, use `GenerateWriteActionRoutine` and `GenerateReadActionRoutine`:

```
GenerateWriteActionRoutine oid
GenerateReadActionRoutine oid
```

For example, if you want to cause MIBMAN to generate a template action routine that writes the value of a scalar object whose OID is `1.3.6.1.4.1.901.999.1.1.1`, use this option:

`GenerateWriteActionRoutine` *`1.3.6.1.4.1.901.999.1.1.1`*

Setting accessor functions

The management API usually controls management variables. It is possible, however, to create variables that are accessed through the API but implemented in application code. You can do this by specifying an accessor function for the variable when it is registered with the API.

If you specify an accessor function, the management API 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 implementing reads and writes to the variable.

To specify that the management variable created for an object should have an accessor function, use `SetAccessorFunction`, which takes this form:

`SetAccessorFunction oid =` *`function`*

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

```
AccessorFunction 1.3.6.1.2.1.1.1=sysDescrFunction
```

Include files

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

`Include` *`filename`*

where *`filename`* specifies the name of the file to include. You can specify any number of include files.

Controlling the names of constants

MIBMAN defines constants in the header file to represent variable identifiers. By default, MIBMAN determines the names of the constants 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`.

To set the prefix for identifiers, use the `IdentifierPrefix` option, which takes this form:

`IdentifierPrefix` *`prefix`*

where:

`prefix` specifies the prefix to use.

For example, using the `IdentifierPrefix MIBII_` option in the configuration file for `RFC1213_MIB` changes the name of `RFC1213_MIB_sysDescr` to `MIBII_sysDescr` (and all other constants, too).

Step 3: Final integration

After you use MIBMAN to generate the C and header fields, use this procedure for final integration:

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 that the MIB variables need.
3. Create and initialize semaphores that protect the variables before the variables are registered with the management API.
4. Provide code that calls `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

This section describes writing action routines for scalar objects and for tables.

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, you need to write action 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. You need to finish these templates to perform the operations.

For example, assume 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 generates 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;
}
```

You need to edit this action routine 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. Store the data 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, which are indicated by the `actionCode` argument. For scalar objects, the important phases are:

- The `SNMP_SET_ACTION` phase, where the new value is written
- 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 you could edit the template 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, assume 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, you can use `GenerateWriteActionRoutine` 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. You need to edit it 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 identifies which row in the table is being read or written. Although the MIB describes the type of index values that are passed, it does not describe how the values are to be used. That information is described in comments.

Some table indexing systems are very complex and create relationships with other tables. You need to modify the action routines that MIBMAN creates for tables 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 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 you need to complete. All columnar objects in a table use the same read and write action routines.

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`. You must implement the table's indexing algorithm to convert the index into one that the management API can use to look up a row. This is indicated by a comment in the template with a "To Do" note.

After the index is converted into a form that the management API can use, 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 as 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 also must 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 the write operation into five phases that are indicated by the value of the `actionCode` argument passed to the write action function.

This table describes the phases:

In this phase	The action routine
`SNMP_SET_RESERVE`	Allocates memory or other resources needed to perform the write operation.
`SNMP_SET_COMMIT`	Copies the value in `info->set To` to the buffers allocated in the `SNMP_SET_RESERVE` phase.
`SNMP_SET_ACTION`	Writes the variable at this point.

In this phase	The action routine
SNMP_SET_FREE	Releases the buffer or other resources previously allocated.
SNMP_SET_UNDO	Restores the variable's original value. The value will be in `info->val`. This phase occurs only if an error is detected.

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 that MIBMAN generates, shown next, 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 tries 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. You 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.

You must implement this code in table write action routines:

- As with the read action routine, you must implement the table's index scheme to encode the SNMP style index `oldIndex`. If the write can change the index of the row, you also need to update `newIndex` with the new index values.
- The console does not necessarily provide values for all the columnar objects in a row. You must implement code to set default values for any the console doesn't provide, and verify that all the required values have been provided.
- You must provide code to verify that the values in the row are valid.
- If rows can be deleted, you 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 update the field value.

SNMP OID and string index values

OIDs and octet string indexes can be encoded in two 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 they change the way in which 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` because 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 comparison 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 the values are not implicit

Sometimes MIB designers are not aware of the difference between implicit and non-implicit index values. They may not expect non-implicit 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 4

This chapter describes the PBuilder utility, which you use to convert HTML Web pages into usable, compilable C source code.

Overview

The Page Builder (PBuilder) 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 you to use an HTML file as an input source file. You can maintain or update an HTML page, rerun the PBuilder utility, and recompile the application program to generate updated images. Working in this way, you 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 ships 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.

Using special tags (described in the next section and in the documentation for the PBuilder utility), you add dynamic content such as option 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 that AWS requires. *Digi 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 you to generate and modify hooks (function stubs) with the present dynamic content inserted.

About the Advanced Web Server Toolkit documentation

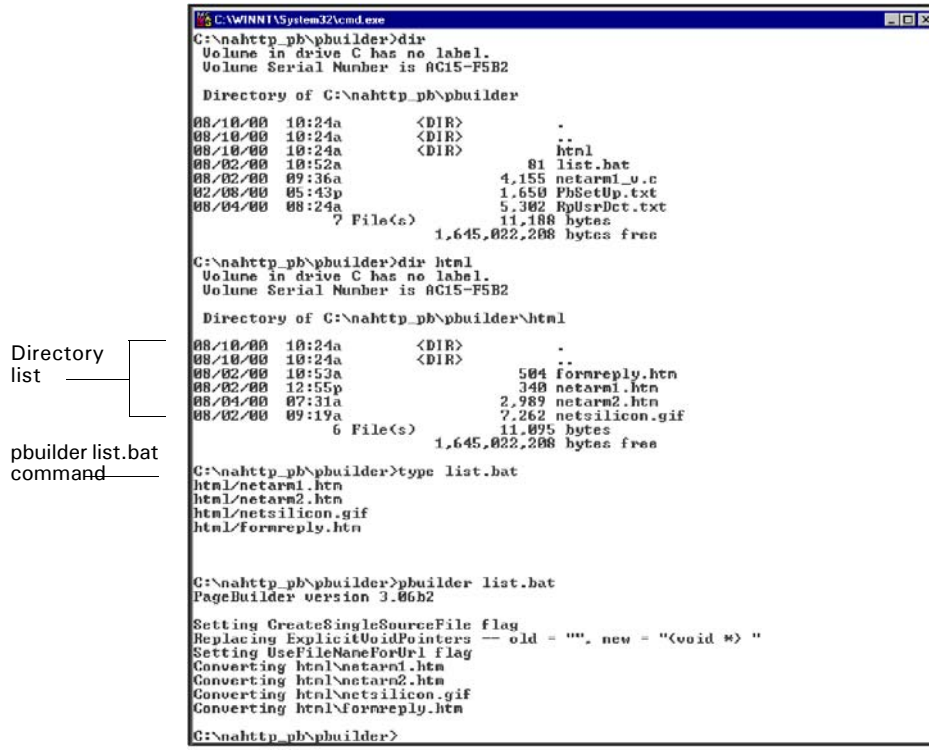
The Advanced Web Server Toolkit documentation, included on the NET+Works CD, describes how to annotate HTML Web content with comment tags to pass dynamic content through the server. The documentation also provides examples.

A portion of the documentation 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 and how the phrase dictionary is used for Web content compression.

Running the PBuilder utility

To run the PBuilder utility from either PBuilder Helper or a DOS prompt, enter: `pbuilder list.bat`.

A window that looks similar to this opens:



The PBuilder Helper window shows a directory list followed by a PBuilder execution and the contents of `list.bat`. The `list.bat` file contains all the Web pages used for the `nahttp_pd` application. The Web page file (that is, `list.bat`) needs either a `.bat` or `.txt` extension.

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`.

You need these additional files to run the PBuilder utility:

- `PbSetUp.txt` — Copy this file from the `nahttp_pd` application directory, and use it to configure the PBuilder utility.
  Do not change this file.
- `RpUsrDct.txt` — Contains definitions for Web content compression and is used to generate the `RpUsrDct.c` and `RpUsrDct.h` files.
  You can update the `RpUsrDct.txt` file 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` file 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 `RpPages.c` file, which contains a structure - `gRpMasterObjectList` - that contains all the application Web pages.

You must compile and link these files for this application:

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

Because `pbuilder\RpUsrDct.h` is required, you need to add the `\pbuilder\` path to the build's include path.

Linking the application with the PBuilder output files

When you build an application, you include the AWS library in the final link of the application. You also need to compile and include three additional files 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.

For examples of overwriting the files, see the `nahttp_pd` or `naficgi` sample applications on the NET+Works 6.3 CD.

security.c file

Using the `security.c` file, you can add up to eight security realms. You can then use the realms to password-protect Web pages.

For more information, see the `nahttp_pd` sample application or the Advanced Web Server Toolkit documentation for the PBuilder utility.

cgi.c and file.c files

You use the `cgi.c` and `file.c` files 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 online help, or the Advanced Web Server Toolkit documentation for the PBuilder utility.

Comment tags

The most important component of the PBuilder utility is the comment tags you insert 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.

The Advanced Web Server Toolkit documentation for the PBuilder utility describes comment tags in detail. Digi strongly recommends that you carefully review the `nahttp_pd` 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 (the HTML between `<!- RpFormInput.... -->` and `<!-- RpEnd -->`) is not used, nor is it required. Digi recommends that you include the HTML, however, to assist when you create Web pages.

Creating Web pages

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

The Advanced Web Server has a built-in way to support a custom interface to system variables. The interface has been adapted to access variables through the management API, allowing you 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 you to create Web pages that can display the current value of variables and prompt users for new values. To create these pages, you 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 comments contain tags that tell AWS to display the variable *`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.

This table describes the tags:

This tag	Tells AWS that
`Name=Username`	The variable is named *`Username`*.
`RpTextType=ASCII`	The variable is an ASCII string.
`RpGetType=Function`	A function has been supplied to read the variable.
`RpGetPtr=GetUsername`	The function is named `GetUsername`.

When AWS encounters this comment, it calls the `getUsername` function, which returns an ASCII string that AWS inserts into the Web page. For more information about using AWS tags, see the Advanced Web Server Toolkit documentation for the PBuilder utility.

AWS normally accesses variables directly through either pointers or functions that you write. 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, with these exceptions:

- You must set the `RpGetType` and `RpSetType` tags to `Custom`.
- 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.

Through modifications to the AWS's customizable routines to access management variables, AWS and the management API have been integrated. So, for example, if the variable *`Username`* were 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

This table shows how AWS data types are mapped to management API data types:

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

To display the values of management variables in Web pages, use the AWS `RpNamedDisplayText` comment tag. The comment tag takes this form:

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

where you replace:

- *`name`* with the name of the management variable to display.
- *`type`* with the AWS type of the variable.

For example, assume that `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

You 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.

RpFormInput

To prompt users for a numeric value or a string, use the `RpFormInput` tag. 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 you replace:

- *`promptType`* with the type of prompt for this input field (text, password, hidden, check box, or option button).
- *`dataType`* with the AWS data type for the variable.
- *`name`* with the name the variable was registered under.
- *`length`* with the maximum length for the variable.
- *`size`* with the size of the input field.

For example, this HTML code prompts for 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 -->
```

RpFormTextAreaBuf

Use `RpFormTextAreaBuf` to prompt users for a string value with a multi-line text box. Use this form:

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

where you replace:

- *`name`* with the variable's name
- *`height`* with the height of the text box
- *`width`* with the width of the text box

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

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

RpFormSingleSelect and RpSingleSelectOption

You can use a select list to prompt for a numeric value to be written into an 8-bit word using the `RpFormSingleSelect` and `RpSingleSelectOption` tags:

- `RpFormSingleSelect` sets up a select list.
- `RpSingleSelectOption` sets up individual items in the select list.

RpFormSingleSelect

`The RpFormSingleSelect` tag has this form:

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

where you replace:

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

RpSingleSelectOption

The `RpSingleSelectOption` tag has this form:

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

where:

- *`text label`* is a label for this option.
- *`numericValue`* is the corresponding numeric value to be assigned to the variable if the user selects this option.

The next example sets up a select list that prompts users to choose a day of the week. The variable *`dayOfTheWeek`* is set to a value between `0` and `6`, depending on which day a 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 you to associate a username and password with a group of Web pages. The combination of the username, password, and list of Web pages is called a *realm*. The AWS requires users to supply a username and password whenever they access any page in the realm. You can create up to eight realms.

Exceptional cases

You may need to write special-purpose code to access management variables. In these cases, you 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

You can configure the MAW module to control:

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

This table shows the default configuration settings:

Setting	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 `mawSetAccessTimeout`function, which is defined as:

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

The `timeout` argument specifies the new timeout value.

Applications also can specify different timeouts for each variable. To do so, use `mawInstallTimeoutFunction` 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`* argument is a pointer to a function supplied by the application that controls the timeouts used for each variable.

mawTimeoutFn type

The type `mawTimeoutFn` is defined as:

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

The *`varName`* argument 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 subscripts to use when the arrays are accessed. You do this by calling the function `mawInstallSubscriptsFunction`, which is defined in this way:

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

The *`appFunction`* argument is a pointer to the application-supplied function that determines the subscripts to use.

mawSubscriptsFn type

The `mawSubscriptsFn` type is defined in this way:

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

where:

- *varName* is a pointer to the variable being accessed.
- *indices* is a pointer to an array of integers that are the current loop indices being used by the HTTP server.
- *dimensions* is a pointer to an array of integers that specify the dimensions of the management variable.
- *numberDimensions* is the number of dimensions the management variable has.
- *htmlType* is 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 characters, the function can 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 *appFunction* argument is a pointer to the application's error handler.

mawErrorFn type

The mawErrorFn type is defined in this way:

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

where:

- *Varname* is a pointer to the variable being accessed.
- *HtmlType* is the data type expected by the HTTP server.
- *Error* is the error condition that is identified.

The function either halts the system or returns a value that the HTTP server can use.

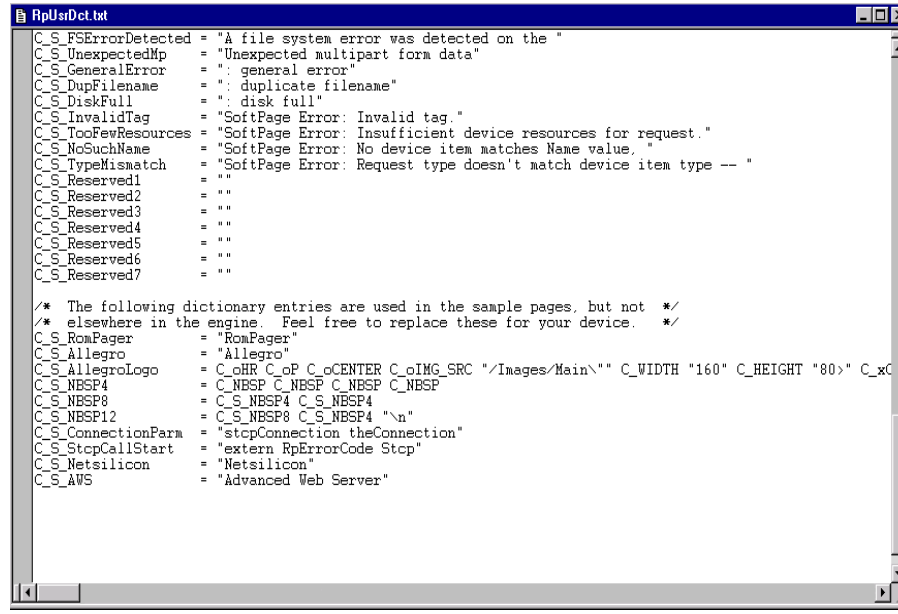
Building the application

The MAW module is built into a special version of the AWS library. Link your application 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 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_pd` application:



You add common phrases in all the application Web pages (for example, a company name that is used several times).

In the sample file, note this definition, which is used several times in the application Web pages:

`C_S_AWS = “Advanced Web Server”`

Search the `\pbuilder\html\netarm1.c` file. The `C_S_AWS` string is used consistently throughout the file.

Maintaining and modifying Web content

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

If a Web page requires a change or a new page, you can either update the HTML, add a new page to the `list.bat` file, or do both. You can add new phrases 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_pd` — This application shows 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 it overwrites the `cgi.c` and `file.c` files to external CGI.

Troubleshooting

CHAPTER 5

This chapter describes how to diagnose some errors you may encounter when you are working with NET+OS. This chapter also describes how to reserialize a development board.

Diagnosing errors

These sections tell you how to diagnose two types of errors:

- Fatal errors
- Unexpected exceptions

Diagnosing a fatal error

Code in the BSP and NET+OS API libraries calls the `customizeErrorHandler` routine when a *fatal error* is encountered. A fatal error is one from which the software cannot recover.

The default version of `customizeErrorHandler` blinks the LEDs on the development board in a pattern that indicates the type of error that occurred.

▶ **To determine where in the code an error occurred:**

1. Stop the program in the debugger.
2. Examine the call stack.

   The call stack lists each function frame on the stack. To go to any of these functions, double-click the function name in the call stack display.
3. To continue execution from the point where the error occurred, set the `naCustomizeErrorHandlerClearToContinue` variable to `0`.

Be aware that because a fatal error has occurred, the results are unpredictable.

Diagnosing an unexpected exception

The `customizeExceptionHandler` routine is called whenever an unexpected exception occurs. This table describes the exceptions:

Exception type	Triggered when
Data abort	Software attempts to access memory that doesn't exist.
Prefetch abort	The processor attempts to fetch an instruction from memory that doesn't exist.
Fast interrupt	The FIQ pin is toggled by hardware, or when internal devices such as the watchdog timer in the NET+ARM are programmed to generate it.

Exception type	Triggered when
Software interrupt	The processor executes a software interrupt (SWI) instruction.
Undefined interrupt	The processor executes an undefined instruction.

The value of the `BSP_HANDLE_UNEXPECTED_EXCEPTION` constant in `bsp.h` controls the default version of `customizeExceptionHandler`. (For details, see the online help.) Usually, `customizeExceptionHandler` either resets the unit or blinks the LEDs in a pattern that indicates the type of exception that occurred. During development, you can continue execution from where the exception occurred.

▶ **To diagnose an unexpected exception:**

1. Put a breakpoint on `customizeExceptionHandler`.
2. When the breakpoint is reached, step into the routine until it sets `customizeExceptionHandlerClearToContinue` to `TRUE`.
3. Set `customizeExceptionHandlerClearToContinue` to `0`.
4. Step through the routine until just before it returns.
5. Switch the debugger display to show assemble instructions.
6. Step through the code assembler instructions one at a time until the processor returns to the source of the exception.

Reserializing a development board

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.

Observing the LEDs

Be aware of the amber and green LEDs whenever you power cycle the development board. The LEDs provide information 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.

Assigning a MAC address to the NET+Works board

Each device on the network must have a unique Ethernet media access controller (MAC) address. The NET+Works development board comes preconfigured with a factory-set MAC address that is printed on a sticker on the board.

The MAC address can be lost if NVRAM is corrupted by an application under test. In such a case, you must restore the MAC address to make sure that the board can communicate over the network. The development board ships with an application written in flash ROM that you can use to restore the MAC address. From the debugger, you also can use any sample application built with the configuration dialog enabled.

▶ **To restore a board's original Ethernet MAC address:**

1. Connect the board to a serial port on your system.
2. Start a HyperTerminal session on the serial port.
3. Power up the board.

   A message similar to this one appears after a brief pause:

```
NET+WORKS Version 6.3
Copyright (c) 2003, NETsilicon, Inc.
PLATFORM: net50bga_a
APPLICATION: Ram-based FTP Server Application
------------------------------------------------------------------
NETWORK INTERFACE PARAMETERS:
IP address on LAN is 1.2.3.4
LAN interface's subnet mask is 255.255.255.0
IP address of default gateway to other networks is 1.2.3.4
HARDWARE PARAMETERS:
Serial channels will use a baud rate of 9600
This board's serial number is N99999999
This board's MAC Address is 00:40:9D:00:43:35
After board is reset, start-up code will wait 5 seconds
Default duplex setting for Ethernet connection: Full Duplex
------------------------------------------------------------------
Press any key in 5 seconds to change these settings.
```

4 Enter the configuration dialog by pressing a key before the timeout expires.

5 At the prompt, enter the system password:

```
Netsilicon
```

6 Enter the values for the IP stack configuration settings and serial port baud rate.

7 At the prompt, enter the Ethernet MAC address that appears on the sticker on the board.

8 Respond to the prompts to set up the remaining configuration settings.

This is a sample dialog:

```
Enter the root password: **********
Reset configuration to default values (Y/N)? Y

For each of the following questions, you can press <Return> to select the
value shown in braces, or you can enter a new value.

NETWORK INTERFACE PARAMETERS:
Should this target obtain IP settings from the network? [N] y

SECURITY PARAMETERS:

Would you like to update the Root Password? [N[

HARDWARE PARAMETERS:

Set the baud rate of Serial channels [9600]?
The new baud rate is 9600
The baud rate will be changed on next power up
Please set the baud rate for your terminal accordingly

Each development board must have a unique serial number
Set the board's serial number [N99999999]? N12345678
The board's new serial number is N12345678

Each development board must have a unique Ethernet MAC address.
Set the board's Ethernet MAC Address [00:40:9D:BA:DB:AD]?
00:40:9D:12:34:56
This board's new Ethernet MAC address is 00:40:9D:12:34:56

How long (in seconds) should CPU delay before starting up [5]?

Normally the board will automatically negotiate with the network hub (or
switch) to determine the Ethernet duplex setting; however some hubs do not
support autonegotiation.
What duplex setting should be used in this case (Full or Half)? [Full
Duplex]

Saving the changes in NV memory...Done.
```

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 you can use to update flash ROM.

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

1. Configure the target development board and in-circuit emulator (ICE).
2. Build the bootloader image.
3. Build the application image, and build the FTP Flash download program from the `naftpapp` application.
4. Send the `rom.bin` file in your platform directory and the `image.bin` file in the `naftpapp/32b` directory
5. Verify the boot ROM image on the target development board.
6. Verify the application.

The next sections provide details about each step in the procedure.

Note: Be aware that the order of the tasks for restoring the contents of flash ROM is important. You *must* do the tasks in the order in which this document presents them.

Step 1: Configure the development board and the MAJIC

▶ **To set up the development board and the MAJIC:**

1. Connect the MAJIC as described in the *NET+Works Quick Install 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 MAJIC.
5. Disable flash on the board. Depending on the board you are using, you might disable flash using either a jumper or a switch. For details, see the jumpers and components guide for the board you are using.
6. Start a HyperTerminal session.

Step 2: Building the bootloader

▶ **To build the bootloader:**

1. Edit `bsp.h` and make sure the configuration settings are correct.

   For information about these settings, see the *NET+Works with Green Hills BSP Porting Guide* and the online help.

2. Build your project as shown in "Appendix A, Using Central Build," in the *NET+Works with Green Hills BSP Porting Guide*.

Step 3: Building the application image and starting naftpapp

▶ **To build the application image and start the naftpapp application:**

1. Prepare the application image.
2. Edit the `appconf.h` file for `naftpapp`, and make sure the application is configured to generate a configuration dialog.

   To generate a dialog, set the constant `APP_DIALOG_PORT` in `appconf.h`.

   For more information about `APP_DIALOG_PORT` in `appconf.h`, see the online help.
3. Rebuild the `naftpapp` application.
4. Start the debugger and load `naftpapp`.
5. Start the application.

   `naftpapp` prompts you with the standard NET+OS configuration dialog box (unless you have disabled this feature).
6. Verify that the network settings are correct, and change them if necessary.

Step 4: Sending rom.bin to the development board

▶ **To send `rom.bin` of the bootloader to the development board:**

1. Open a command shell.
2. Change to this directory:

   `C:/netos63_ghs/src/bsp/your-platform`

   where you replace `your_platform` with the name of your platform.

3. To start the Windows FTP client, enter this command and press Enter:

   `FTP a.b.c.d`

   and press Enter.

   where *a.b.c.d* is your unit's IP address.
4. When you are prompted for a username, enter:

   `(none)`
5. To select binary mode transfer, enter:

   `bin`
6. Enter this command:

   `put rom.bin`
7. When the transfer is complete, enter:

   `quit`
8. Exit from the debugger.

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

At this point, the bootloader has been written into the boot sector of flash. Now you need to write the application into flash.

▶ **To write an application into flash:**

1. Build the application you want to write into flash.
2. Restart the `naftpapp` application in the debugger.
3. Change to this directory:

   `C:/netos63_ghs/src/examples/naftpapp/32b`
4. To start the Windows FTP client, type this command and press Enter:

   `FTP a.b.c.d`

   where *a.b.c.d* is your unit's IP address.
5. When you are prompted for a username, enter:

   `(none)`
6. To select binary mode transfer, enter:

   `bin`

7 Enter this command:

```
put image.bin
```

8 When the transfer is complete, enter:

```
quit
```

9 Exit from the debugger.