Porting the ARM SNMP Agent

Version 1

Programmer's Guide

ARM

ARM DUI 0120A

Release Information

The following changes have been made to this document.

Change history

Date	Issue	Change
January 2000	A	First release

Proprietary Notice

Contents
Programmer's Guide

Preface

This preface introduces the ARM SNMP Agent porting procedure. It contains the following sections:

About this book

This book is provided with the ARM SNMP Agent software.

It is assumed that you have the ARM SNMP Agent porting sources available as a reference. It is also assumed that you have access to programmer guides for C and ARM assembly language.

Intended audience

This Programmer's Guide is written for experienced embedded systems programmers, with a general understanding of what an SNMP Agent does. It is written for those programmers who want to use the ARM SNMP Agent in their product.

Using this book

This book is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an introduction to the SNMP Agent, and to learn about the demonstration program and system requirements for the SNMP Agent.

Chapter 2 *MIB Compiler*

Read this chapter for an overview of the MIB compiler, and for details on how to build and use the compiler.

Chapter 3 *Porting the SNMP Agent*

Read this chapter to learn how to port the ARM SNMP Agent, step-by-step, to an embedded system.

Chapter 4 *Function Descriptions*

Read this chapter for a description of the functions and data items used to interface to the SNMP Agent core.

Appendix A *SNMP ARP Table Interface*

Read this appendix to see a heavily annotated example of the `atEntry` table, a complete example of using an `access_method` function (see *Custom SET operations* on page 3-26).

Appendix B *Building the Demonstration Program*

Read this appendix for complete instructions on building the demonstration SNMP Agent shipped with ARM SNMP.

Typographical conventions

The following typographical conventions are used in this book:

`typewriter` Denotes text that may be entered at the keyboard, such as commands, file and program names, and source code.

`type`writer Denotes a permitted abbreviation for a command or option. The underlined text may be entered instead of the full command or option name.

`typewriter italic`
Denotes arguments to commands and functions where the argument is to be replaced by a specific value.

italic Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.

bold Highlights interface elements, such as menu names and buttons. Also used for terms in descriptive lists, where appropriate.

`typewriter bold`
Denotes language keywords when used outside example code and ARM processor signal names.

Further reading

This section lists publications from both ARM Limited and third parties that provide additional information on SNMP.

ARM periodically provides updates and corrections to its documentation. See `http://www.arm.com` for current errata sheets and addenda.

See also the ARM Frequently Asked Questions list at:
`http://www.arm.com/DevSupp/Sales+Support/faq.html`

ARM publications

This book contains reference information that is specific to the ARM SNMP Agent. For additional information, refer to the following ARM publications:

- *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041)
- *ARM Software Development Toolkit User Guide* (ARM DUI 0040)
- *Porting TCP/IP Programmer's Guide* (ARM DUI 0079).

Other publications

For other reference information relating to the ARM SNMP Agent, please refer to the following:

- Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language*, 2nd Edition, 1988, Prentice-Hall (ISBN 0-13-110370-8).
- Rose, Marshall T., *The Simple Book; An Introduction To Internet Management, Revised*, 2nd Edition, 1995, Prentice-Hall (ISBN 0-13-451659-1).
- RFC 1155, McCloghrie, K., Rose, M., "Structure and Identification of Management Information for TCP/IP-based Internets", May 1990.
- RFC 1157, Case, J., Davin, J., Fedor, M., Schoffstall, M., "A Simple Network Management Protocol (SNMP)", May 1990.
- RFC 1212, McCloghrie, K., Rose, M., "Concise MIB Definitions", March 1991.
- RFC 1213, McCloghrie, K., Rose, M., “Management Information Base for Network Management of TCP/IP-based internets: MIB-II", March 1991.
- RFC 1700, Postel, J., Reynolds, J., "Assigned Numbers", October 1994.
- RFC 2325, Slavitch, M., “Definitions of Managed Objects for Drip-Type Heated Beverage Hardware Devices using SMIv2”, April 1998.

Feedback

ARM Limited welcomes feedback on both the ARM SNMP Agent, and its documentation.

Feedback on the ARM SNMP Agent

If you have any problems with the ARM SNMP Agent, please contact your supplier. To help them provide a rapid and useful response, please give:

- details of the release you are using
- details of the platform you are running on, such as the hardware platform, operating system type and version
- a small standalone sample of code that reproduces the problem
- a clear explanation of what you expected to happen, and what actually happened
- the commands you used, including any command-line options
- sample output illustrating the problem
- the version string of the tool, including the version number and date.

Feedback on this book

If you have any comments on this book, please send email to `errata@arm.com` giving:

- the document title
- the document number
- the page number(s) to which your comments apply
- a concise explanation of the problem.

General suggestions for additions and improvements are also welcome.

Chapter 1
Introduction

This chapter introduces the SNMP Agent, and describes the demonstration program and system requirements for the SNMP Agent. It contains the following sections:

1.1 About the ARM SNMP Agent

This technical reference is provided with the ARM portable *Simple Network Management Protocol* (SNMP) sources. The purpose of this document is to provide enough information to enable a moderately experienced C programmer, with a reasonable understanding of SNMP, to integrate the ARM SNMP Agent into a product. It is assumed you have experience with networking code, especially *Transmission Control Protocol/Internet Protocol* (TCP/IP).

1.1.1 Demonstration program

It is assumed you have the ARM SNMP Agent demonstration program available as a reference. This is provided as part of the SNMP Agent source code release. Currently, the demonstration program is shipped for an ARM Development Board (PID7T), using ARM's *Berkeley System Distribution* (BSD) UNIX-derived ARM TCP/IP that compiles with the *ARM Software Development Toolkit* (SDT) version 2.50. The ARM Development Board (PID7T), SDT 2.50, and TCP/IP can be purchased from ARM. See the web site `http://www.arm.com`.

SNMP demonstration program

The demonstration program implements an SNMP agent that supports the *Management Information Base*-II (MIB) System and SNMP groups. See Appendix B *Building the Demonstration Program* for complete details on building and running the demonstration program.

1.2 Terms and conventions

In this document, the following terms are used:

Agent When used without other qualification, means the ARM SNMP Agent code as ported to an embedded system.

ASN.1 Abstract Syntax Notation 1. Refers to a subset of the ISO/ITU-T standard, as described in *RFC 1155*.

End user Refers to the person who ultimately uses your product.

Sockets Refers to the TCP/IP *Application Program Interface* (API) developed for UNIX at the University of California, Berkeley.

The agent is delivered with example implementation notes for Sockets because many embedded systems already have sockets. A copy of the Sockets API documentation is available from ARM upon request.

Stack Means the TCP/IP and related code, as ported to an embedded system.

System Refers to the embedded system.

You Refers to the user or engineer who is porting the server.

Conventions used throughout the document, such as the use of bold or italic font, are explained in *Typographical conventions* on page Preface-vii.

1.3 System requirements

To port the ARM SNMP Agent to another environment, you must have the following resources available in the target environment:

- sufficient RAM (see Table 1-1 on page 1-5 for approximate values)
- a networking stack that supports *User Datagram Protocol* (UDP).

Note

An SNMP Agent-compatible TCP/IP stack is available, and this has been optimized for the ARM architecture. Contact your ARM supplier for details. Your target platform may come with a suitable protocol stack.

1.3.1 MIB Variables

The MIB variables that you define and implement affect the system requirements. In general, as more MIB variables are defined, more memory is required for both code and data. Variables defined in a table, as part of an ASN.1 SEQUENCE, generally require more code space and CPU power than nontabular variables. Code written to maintain the variables in structures produced by the ARM MIB Compiler `-h` option also tends to be more efficient than code adapted to an existing system. Systems that maintain the implemented variables in hashed or ordered tables also tend to be faster than those that do not, but they might require more memory to manage the hashed data structures.

There is no easy way to determine the exact memory sizes required. However, you can get a general idea by examining the demonstration program. Some figures are given in Table 1-1 on page 1-5 for a MIB-II demonstration, which is part of the standard package.

Note

The figures in Table 1-1 on page 1-5 are subject to change without notice, but are current at the date of publication.

Table 1-1 Memory sizes of the MIB-II demonstration

Use	Bytes
SNMP Agent ROM space	7956
SNMP Agent RAM space	2760
SNMP variables ROM space (includes `var_` routine code)	13552
SNMP variables RAM space (used by `var_` routines)	80

Note

When the demonstration is run, it also allocates six small packet buffers (128 bytes each) and three large ones (600 bytes each) dynamically using `malloc()`. The memory used by these packets should be added to the figures shown in the configuration in Table 1-1.

The program is compiled using ARM SDT 2.50, for the ARM7TDMI processor in Thumb state, using options to optimize code size (rather than execution speed). It implements MIB-II as described in *RFC 1213*. It does not have a TCP or *Exterior Gateway Protocol* (EGP) layer, and therefore omits the group for this protocol as allowed by the RFC.

You can cut the amount of space used by SNMP code and data by deleting groups of variables from the MIB.

The sizes shown in Table 1-1 will change when compiled for other ARM cores. The sizes generally increase when compiled for ARM state, or when optimized for speed rather than code size.

1.3.2 Processing power

Processing power is another resource for which exact requirements are difficult to quantify. The SNMP Agent requires only that it can reply to the requests of an SNMP management station before the timer runs out. If the station sends only one request per second, and allows one second for the reply, any processor will be adequate. However, the stations sometimes send packets in bursts, and the CPU of the SNMP Agent has tasks other than SNMP to perform. As discussed in *MIB Variables* on page 1-4, the number and type of MIB variables implemented also has a major effect on CPU requirements, as the table of variables has to be searched for each received request.

To assess processing power requirements, it is recommended that you first compare requirements with those of similar systems. For embedded agents, the ARM SNMP demonstration is intended to provide a starting point. You can add and delete variables quickly using the MIB Compiler, and then test them on the ARM Development Board (PID7T). You can run other processes simultaneously with the agent to determine the effect of heavy SNMP work on the performance of the system.

Chapter 2
MIB Compiler

This chapter gives an overview of the MIB Compiler. It also provides details on how it is built, and how to use it. It contains the following sections:

2.1 About the MIB Compiler

The ARM MIB Compiler is a program that takes a MIB file written in RFC 1155-compliant ASN.1 notation, containing the specification of the variables to be managed, and produces one or more of these output files:

- Files containing C language source code files, that are useful for beginning the implementation of the SNMP MIB variables in the SNMP Agent. These files contain *skeleton* routines that you need to fill in.
- A *variables* file that contains a table that links the MIB variables to the skeleton routines.
- `.h` header files with the function prototypes and definitions for the above `.c` files.
- A *numbers* file, suitable for describing the MIB variables to an SNMP management station.

Filling in the skeleton routines (also called *stub* routines) comprises a major portion of the work involved in porting and maintaining your SNMP Agent. The routines can be quite complex, so a major portion of this chapter and the next (Chapter 3 *Porting the SNMP Agent*) describes these routines. Generally, the most difficult part of an SNMP implementation is understanding what these routines do, how they index Object IDs, and how they access variables in tables.

2.2 Building the MIB Compiler

The MIB Compiler is provided in source form. You will need to build `mibcomp` using the native C compiler of your system. The sources consist of three `.c` files and one `.h` header file:

`main.c` Is the system-dependent front end with the `main()` routine.

`parse.c` Reads MIB files.

`parse.h` Contains the defines for `mibcomp`.

`tree.c` Writes the output files.

These files should compile with little or no modification on most machines. They include standard C library calls and header files. Makefiles are provided for Solaris, GNU C, the Microsoft `nmake` facility, and ARMCC (for use with the ARMulator).

Note

The MIB Compiler is used regularly during the development of the SNMP Agent as MIB variables are changed. Therefore, you must store the `mibcomp` executable in a directory where it can be invoked by the makefiles that build your embedded system.

2.3 Usage

The MIB Compiler command-line format is as follows:

`mibcomp -i` *`mibfile.mib`* `[`*`mibfile2.mib`* … `] [-[chnvq]]`

where:

`-i` *`mibfile.mib`* …
: Is a space-separated list of text files containing the ASN.1 descriptions of the MIBs to be compiled.

`-c`
: Outputs a `.c` C source file that contains skeleton C routines for parsing the variables in each group within the MIB.

`-h`
: Produces a `.h` C header file that contains defines and function prototypes for the `.c` file that is produced by the `-c` option.

`-n`
: Outputs a numbers file for each MIB, containing the numeric *Object ID* (OID) and symbolic name and type for each variable within the MIB.

`-v`
: Outputs an `snmpvars.c` file, containing the SNMP `variables` table (see *Variables structure* on page 2-7). The table contains all the variables from all the MIB files.

`-q`
: Reduces the amount of detailed output from the MIB Compiler, making it more suitable for use in script files or makefiles.

The MIB Compiler also supports the following options that might be useful when working with other environments or tools:

`-p`
: Causes the C file produced by the MIB Compiler to be pre-ANSI, that is, with simple prototypes.

`-f`
: Causes the C file produced by the MIB Compiler to use **`far`** pointer references, for use with Intel x86-segmented environments.

2.4 Input

The MIB Compiler takes, as input, one or more MIB definition files as described in *RFC 1155*. Several samples of these files are shipped with the ARM demonstration program. These sample files were created using a simple text editor to edit out the non-ASN.1 portions for the RFCs from which each MIB came. To be left with only ASN.1 that is suitable for input to the MIB compiler, you must delete the following:

- all text preceding the `DEFINITIONS ::= BEGIN` line
- all text following the `END` line
- all page break text (footer, page break, and header).

If you want to define your own MIBs, the two most important decisions to make are:

- which variables you want to manage
- how to organize these variables.

You then need to write the ASN.1 text to describe them, which is a straightforward process. After the MIB has been written in RFC 1155-compliant ASN.1, you can compile and implement it like any standard MIB.

2.5 Output

As discussed in *About the MIB Compiler* on page 2-2, the compiler will produce a `.c` file, a `.h` header file, and a numbers file. Because these files are regenerated whenever the MIB Compiler is run (that is, whenever the MIB is changed), you must not edit them manually. The files are:

source`.c` This C source file contains stub routines for accessing MIB variables. These are intended to be copied to your own source files and completed there.

header`.h` This C header file includes prototypes for the C stub routines, and token definitions for the MIB variables. They also contain suggested data structures to hold the variables for each group or sequence in the MIB.

`snmpvars.c`
This file contains the table that maps the MIB variables and groups to the stub routines that retrieve and set your variables. This file is produced whenever the MIB Compiler is run with the `-v` flag.

The actual names for both *`header`* and *`source`* will be determined by the name given on the `DEFINITIONS` line in the last input MIB file. For example, the `DEFINITIONS` line in *RFC 1213* is as follows:

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

This results in the file `RFC1213.c`. The names of the other output files are similarly determined (`RFC1213.h` and `RFC1213.num`, in this case).

The `snmpvars.c` file contains the variables structure that the SNMP Agent code uses to associate a `.c` routine with individual variables. For more details on the variables structure, see *Variables structure* on page 2-7.

The `.c` file contains stubs for the C routines prototyped in the `.h` header file. These routines are framed and commented. However, they are only empty frames with no internal code. The areas where code needs to be added to actually implement the variables are flagged with the text `TODO`. Part of implementing a new MIB requires that you:

1. Copy these stubs into your own source files.
2. Replace the `TODO` lines with C code that performs the intended `SET` or `GET` operation, and returns the correct values.

See *GET operations and scalar variables* on page 3-15, *GETNEXTs and indexes* on page 3-22, and *Custom SET operations* on page 3-26 for more details.

2.5.1 Variables structure

Discussion of the MIB Compiler output in this section assumes an understanding of certain SNMP concepts, such as:

- lexicographic ordering
- Object IDs and their components
- MIB groups
- ASN.1 SEQUENCEs.

It is recommended that you become familiar with all of these concepts. One book you might find useful is *The Simple Book; An Introduction To Internet Management, Revised*.

The internal routines of the ARM SNMP Agent access the MIB variables using the `variables` table which the MIB Compiler produces in the `snmpvars.c` file. The table, named `variables`, is an array of variable structures, reproduced below as currently defined. You should refer to the source file `snmp_var.h` as the final authority.

```
struct variable {
    oid      name[DEF_VARLEN]; /* obj. identifier of variable */
    u_char   namelen;          /* length of above */
    char     type;             /* type of variable, INTEGER or
                               /* (octet) STRING */
    u_char   magic;            /* passed to func. as a hint */
    u_short  acl;              /* access control list for */
                               /* variable */
    u_char   *(*findVar)();    /* func. that finds variable */
};
```

One of these structures is defined for each accessible variable in the input MIBs. Example 2-1 shows the example compiler output from the C file for the beginning of the MIB-II system group:

Example 2-1

```
struct variable variables[] = {
        {{1,3,6,1,2,1,1,1,0}, 9, STRING,    SYSDESCR,    RONLY,  var_system },
        {{1,3,6,1,2,1,1,2,0}, 9, OBJID,     SYSOBJECTID, RONLY,  var_system },
        {{1,3,6,1,2,1,1,3,0}, 9, TIMETICKS, SYSUPTIME,   RONLY,  var_system },
        {{1,3,6,1,2,1,1,4,0}, 9, STRING,    SYSCONTACT,  RWRITE, var_system },
…
```

The entry fields in the `variables` table are as follows, presented in order:

- The Object ID of the variable, stored as an array of unsigned values.
- The length of the Object ID (in this case, they all consist of nine values, but this length will vary).
- The ASN.1 type of the variable.
- A token that is passed to the access routine so it can determine which of the variables in its group or sequence to act on. These *magic* tokens are also generated by the MIB Compiler, and are unique within their group or sequence. They usually start at 0, and then increment throughout the group or sequence by 4s. This is so they can be used to index byte-wise into the suggested `Group` structures produced for each MIB group in the `.h` header file. For more details, see *Suggested data structures* on page 3-19.
- A field that controls access to the variable. This access can be either read-only or read/write.
- A pointer to a routine used to access the group or sequence of which the variable is a member, `var_system()` in this case, because all of these variables are members of the system group.

  The `var_system()` function is one of the functions prototyped and stubbed in by the MIB compiler. One of these routine stubs is generated by the compiler for each group or sequence in the input MIBs.

The entries in the `variables` table are ordered lexicographically by Object ID so the SNMP Agent code can do fast lookups on the table to find matching variables or appropriate `GETNEXT` entries.

2.6 Updating MIBs

During the course of most agent implementations, the MIB is changed. This is either because the definitions in your private enterprise MIBs have evolved, or because the industry standard MIBs have been updated. When this happens, you must edit the existing C code to reflect the changes.

You should rerun the MIB Compiler to produce new variables, `.c`, and `.h` files. It is recommended that you make the MIB sources (ASN.1 text files) dependencies in your makefile, and include a build rule to run the MIB Compiler.

If you delete one or more variables, the system will compile and operate as before (except without the deleted variable(s)). It is recommended that you delete or comment out the stub-derived code that implemented the operations of the variable. If you change the meaning of a variable, you will have to change the stub code accordingly.

If you create new variables, you must copy the new stubs from the output `.c` file into your implementation file, and modify them to implement the required functionality.

Chapter 3
Porting the SNMP Agent

This chapter outlines what you need to do, step-by-step, to port the ARM SNMP Agent to an embedded system. It contains the following sections:

3.1 Setting up your source tree

The ARM SNMP Agent source files belong to one of four groups, as described in this section:

- *Core files* on page 3-3
- *Port files* on page 3-3
- *MIB Compiler output files* on page 3-4
- *Your project files* on page 3-4.

The structure shown in Figure 3-1 is for a typical project using the ARM SNMP Agent with the ARM UDP/IP stack.

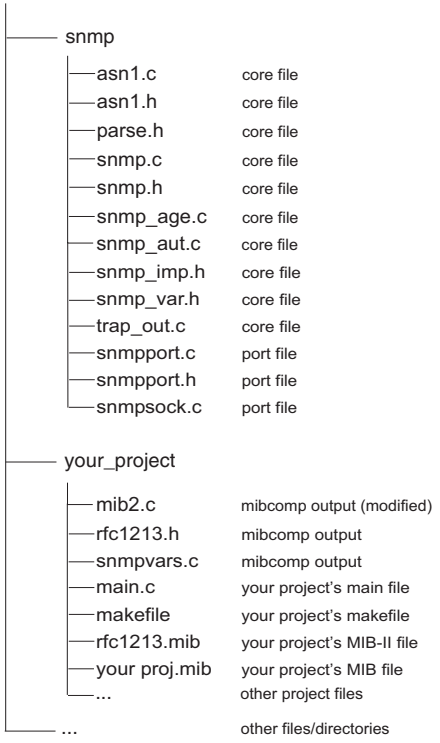


Figure 3-1 Typical directory structure of source files

3.1.1 Core files

The ARM SNMP Agent core files are:

`asn1.c` Contains routines to parse and build ASN.1-encoded data.

`snmp.c` Contains routines to build SNMP *Protocol Data Unit*s (PDU) and variables.

`snmp_age.c`
Contains the SNMP Agent interface routines.

`snmp_aut.c`
Contains authorization routines used to process community strings.

`trap_out.c`
Contains routines used to build and send *trap*s from the SNMP Agent.

Note

These core files and their associated header files should not have to be modified for a port. If you think they need to be changed, please contact ARM technical support before changing them.

3.1.2 Port files

The ARM SNMP Agent port files are:

`snmpport.c`
Contains routines to interface the SNMP Agent to the ARM low-overhead UDP API of the ARM TCP/IP stack.

`snmpsock.c`
Contains routines to interface the SNMP Agent to the ARM TCP/IP stack using the Berkeley sockets API.

You must include either `snmpport.c` or `snmpsock.c` in your project, but not both.

3.1.3 MIB Compiler output files

The MIB Compiler is used to parse the MIB files and generate C stub and header files, and the `snmpvars.c` file that contains the SNMP variables table. Usually, you will not need to modify the C header file or `snmpvars.c` after the MIB compiler has created them. The C stubs file must be renamed (in Figure 3-1 on page 3-2, it has been renamed to `mib2.c`), or the stubs must be copied to another source file, so that it will not be overwritten if the MIB Compiler is used again. It is likely that you will make extensive changes to these stub routines because they implement the interface between the SNMP Agent variables and your embedded system.

3.1.4 Your project files

The remaining files implement the rest of your project and manage your build environment. You also need access to your network system stack routines, either as a library or in source form.

3.2 The target system

The ARM SNMP Agent is suitable for use either with an embedded *Real-time Operating System* (RTOS), or in a polled or superloop environment.

If you are using an RTOS, you will typically establish a separate task for the SNMP Agent, and arrange for the network task to pass incoming SNMP datagram messages to the SNMP task by way of a mailbox or other message-passing facility. The SNMP task processes these datagrams by stripping the network header layers, such as IP or UDP headers, and passing the resultant ASN.1 data to the `snmp_agt_parse()` function for processing. The `snmp_agt_parse()` function returns a response packet to be passed to the network task for transmission back to the management station. It is usual for the network stack and the SNMP Agent to be implemented as separate tasks in this environment.

If you are using ARM TCP/IP as your network transport, it is possible to integrate the SNMP Agent directly with the low-overhead UDP API. This allows you to have both the SNMP Agent and the network stack running together as one task, with the UDP stack making direct calls to the SNMP Agent, and the SNMP Agent directly calling the UDP stack with the responses. This works well with both the superloop and RTOS-task style of implementation discussed in *Porting TCP/IP Programmer's Guide*.

3.3 Porting procedure

The ultimate purpose of any SNMP Agent is to implement a set of MIB variables which can be read or set by network management applications. ARM provides the SNMP layer. The porting programmer must join the SNMP code to a lower transport layer to send and receive messages, and to an upper layer of code that resolves the individual MIB variables into actual values or operations. You can accomplish this with the ARM tools, as follows:

1. Determine what MIB variables are to be used. This involves selecting standard MIBs from the RFCs and, possibly, writing proprietary MIB extensions.
2. Process the MIBs with the MIB Compiler to create skeletal stub C code files containing variable routines for the port. This is discussed in Chapter 2 *MIB Compiler*.
3. Compile the SNMP *core* source files (listed in *Setting up your source tree* on page 3-2), and the TCP/IP source files, if needed, and link with the target system and stub routines. You can test this with a call to `snmp_agt_parse()`.
4. Modify the port files and complete the C code in the stub routines.
5. Compile the new variable routines and link them with the core SNMP files to create an executable image for the new target system. Test and debug the code.

The work involved in porting the ARM agent to a new system involves:

- attaching the SNMP code to the transport layer API
- implementing the stub routines.

These steps are represented by the middle box and top box, respectively, in Figure 3-2.

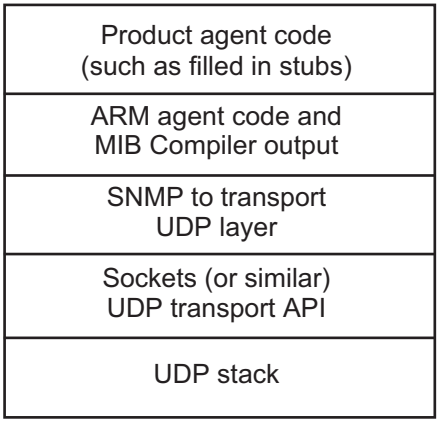


Figure 3-2 Steps for porting the ARM SNMP Agent

3.3.1 SNMP Agent software interface

The ARM SNMP Agent core files interface to the UDP stack and the MIB variables using the following routines and data structures:

- *Functions and macros to implement*
- *Functions to call* on page 3-8
- *Data objects to implement* on page 3-8.

Full details of their syntax can be found in Chapter 4 *Function Descriptions*.

Functions and macros to implement

You need to implement the following as functions in `snmpport.c` , or as macros in `snmpport.h` (or `ipport.h`, if you are using the ARM TCP/IP stack):

`send_trap_udp()`
: This function is called by the SNMP Agent when it has constructed an SNMP trap message that it needs to send.

`ip_myaddr()`
: This is called by the SNMP Agent when it needs to ascertain the IP address of the piece of equipment on which it is running.

`GetUptime()`
: This function is called by the SNMP Agent when it needs to know how long the system has been up and running.

`MEMCPY()` This macro copies bytes from one location to another.

`MEMMOVE()` This macro moves bytes from one location to another. It must be able to allow for overlapping memory areas without corrupting the data.

`dtrap()` This function is used to trap to the debugger when the SNMP Agent detects a problem.

`ntohl()`,`ntohs()`,`htonl()`,`htons()`
: These are used for byte swapping on little-endian systems.

`dprintf()` This function is used by the SNMP Agent for printing debugging information.

`SNMPERROR()`
: This macro is used by the SNMP Agent for reporting errors.

`snmp_init()`

Although this is not actually called by the SNMP Agent, you need to implement this function and arrange for it to be called before any SNMP UDP datagrams are passed to the SNMP Agent. This function performs such tasks as opening a UDP endpoint and binding it to the SNMP port (usually port 161). If your system is configured to support SNMP trap messages, this function must also allocate a buffer in which trap messages can be constructed by the agent.

Functions to call

When porting the ARM SNMP Agent, you will need to make calls to the following functions, usually from code within the `snmpport.c` file:

`snmp_agt_parse()`

Is called when the UDP stack has received a datagram (usually on port 161) to be processed by the SNMP Agent.

`snmp_trap()`

Is called when an SNMP trap message should be constructed and sent, as at system startup, for example, when a *Cold Start trap* is sent.

Data objects to implement

The SNMP Agent also requires the following data objects:

Table of MIB variables

```
int num_variables;
struct variable variables[];
```

This data, as produced by the MIB Compiler in `snmpvars.c`, need not be modified.

Table of SNMP community strings and access permissions

```
int num_communities;
struct communityID communities[];
```

This data structure is usually implemented in `snmpport.c`. You need to either include code in your `main()` routine, or alter the default strings included in this table at compile time to change the SNMP communities to which the SNMP Agent will respond. The demonstration program shows one way of setting these strings from values held in *Non Volatile Random Access Memory* (NVRAM).

system.sysObjectID.0 value

```
int sys_id_len;
oid sys_id[];
```

This value must be set for the piece of equipment you are using. The value is usually part of the `.iso.org.dod.internet.private.enterprises` (`.1.3.6.1.4.1`) tree. You need to obtain a *Private Enterprise Number* from the *Internet Assigned Numbers Authority* (IANA). Please contact IANA for more information and an application form.

SNMP usage statistics

```
struct snmp_mib SnmpMib
```

This structure is usually defined in `snmpport.c`. It is only needed if `MIB_COUNTERS` is defined. In this case, it can be used by the SNMP Agent to keep track of the numbers of various SNMP requests/responses received. It can also be used to implement the SNMP group within MIB-II (refer to the implementation of `var_snmp()` in the demonstration program for an example of this).

3.3.2 Port-dependent files

Most of the work involved with porting SNMP involves modifying or recoding the routines, definitions, and variables in the *port* files. These files are:

- `nptypes.h`
- `snmpport.c`
- `snmpport.h`
- MIB Compiler output files.

The header file `nptypes.h` defines a set of variable types used by the SNMP code. This file is similar to `types.h` in UNIX systems. On most systems, the sample `nptypes.h` header file from the demonstration program works. On some systems, all of these types may already be defined by other system header files. A few systems might require that you edit the `nptypes.h` file to enable SNMP to work properly.

The header file `snmpport.h` contains definitions of a variety of SNMP limits (such as the longest datagram size and the longest `ObjectId` size) and definitions which bind the agent to the host system. This file is included in every C source file in the agent sources. Definitions of the protocol stack API and system library prototypes, for example, must be included here to ensure they are applied uniformly across the SNMP Agent module.

The MIB Compiler output files are described in *Output* on page 2-6. The majority of the work in implementing an SNMP Agent involves writing code to fill in the stubs.

The SNMP Agent sources use a variety of definitions and C library calls. Where possible, ANSI standards are used. However, some of the portability functions are adapted from bespoke standards set by *Berkeley Software Distribution* (BSD) or PC-DOS custom. Because not all embedded system development environments support all of these standards, they are described in detail in this section. Working examples of these are included in the demonstration program.

`TRUE`, `FALSE`, and `NULL` must be defined in the `snmpport.h` header file. To do this, it is recommended that you include the standard C library file `stdio.h` inside `snmpport.h`. Use the code in Example 3-1, which works in most C environments, if `stdio.h` is impractical to use, or is missing.

Example 3-1

```
#ifndef TRUE
#define TRUE -1
#define FALSE 0
#endif
#ifndef NULL
#define NULL (void*)0
#endif
```

Four common macros are used for performing byte-order conversions between different CPU architecture types:

- `htons()`
- `htonl()`
- `ntohs()`
- `ntohl()`.

They can be either macros or functions. They accept 16-bit and 32-bit quantities as shown, and convert them from network format (big-endian) to the format of the local CPU. Most IP stacks already have these byte-ordering macros defined. If this is the case, you must attempt to find the existing include file which defines them, and use this rather than duplicate them. The information in Example 3-2 on page 3-11 and Example 3-3 on page 3-11 is provided in case these macros are not already available.

For big-endian systems, these can simply return the variable passed, as in Example 3-2:

Example 3-2

```
#define htonl(long_var)  (long_var)
#define htons(short_var) (short_var)
#define ntohl(long_var)  (long_var)
#define ntohs(short_var) (short_var)
```

Little-endian systems require the byte order to be swapped. You can use the `lswap()` and `bswap()` functions provided with the ARM demonstration program, as shown in Example 3-3.

Example 3-3

```
#define htonl(long_var)  lswap(long_var)
#define htons(short_var) bswap(short_var)
#define ntohl(long_var)  lswap(long_var)
#define ntohs(short_var) bswap(short_var)
```

The `dtrap()` and `SNMPERROR()` functions are debugging aids. The `dtrap()` function is called by the SNMP code whenever it detects an event which should not have occurred. The intention is for the `dtrap()` function or macro to attempt to trap to whatever debugger is in use by the programmer. It must be treated as an embedded breakpoint.

The SNMP code generally continues executing after calling `dtrap()`, but calls to the `dtrap()` function usually indicate that something is wrong with the SNMP port.

Warning

No product based on this code should be shipped until all calls to `dtrap()` have been eliminated. When you are ready to ship code, you can redefine the `dtrap()` functions to a null function to slightly reduce code size.

For each port, you must define a data type for the individual subcomponents of an SNMP Object ID (*Sub-Id*). This type is named `oid`, and is used throughout the SNMP code. Generally, this is an unsigned 32-bit number. However, applications that only have a small space into which they must fit, can shrink the variables table produced by the MIB Compiler (and therefore save considerable static data memory) by making this 16 bits, or possibly eight bits. There must also be a definition of the maximum value which can be placed in a Sub-Id. A typical definition is as follows:

```
typedef unsigned long  oid;
#define MAX_SUBID      0x7fffffff
```

Note

Throughout the SNMP sources, the term `Oid` is used to refer to both a Sub-Id, as above, or a complete SNMP Object ID.

Compile-time size limits

Because the agent is written without any internal calls to `malloc()` or `free()`, and needs to save small amounts of dynamic data somewhere, it has several compile-time size limits. These are described in this section, with recommended settings and examples.

The SNMP Agent limits the maximum size, in bytes, required to hold an encoded SNMP Object ID. The macro for this is `MAX_OID_LEN`. The real limit required is determined by your MIB. The value 64, assigned in this example, is sufficient for most applications.

```
#define MAX_OID_LEN   64    /* max length in bytes of encoded */
                            /* oid */
```

There is a limit on the maximum size to which a community string can grow. This example sets it at 32 bytes:

```
#define MAX_COMMUNITY_SIZE   32   /* max length in bytes of a */
                                  /* community */
```

This imposes a size limit on the various strings in the MIB-II System group. The port can be rewritten to dynamically allocate memory for arbitrarily large strings:

```
#define SYS_STRING_MAX   256      /* max length of sys group */
                                  /* strings */
```

It is assumed that every company which ships an SNMP-enabled product has an SNMP Enterprise ID. This is a number assigned by IANA which uniquely identifies each vendor of SNMP-managed devices. You must obtain such a number before you ship your product. Occasionally, the contacts for reaching IANA change, but can be obtained from the latest *Assigned Numbers* RFC (*RFC 1700* at the time of publication). To send traps or implement any basic MIB, the SNMP code will require an Enterprise ID. ARM customers are permitted to use the ARM Enterprise ID during product development, but they should obtain and recompile with their own identification prior to *First Customer Ship* (FCS). The definition is:

```
#define ENTERPRISE   4128    /* ARM enterprise Number */
```

The ARM SNMP layer maintains packet counters as defined by MIB-II (*RFC 1213*). For ports that do not require these counters, a small amount of space can be saved by omitting them. The counters are enabled with the following define:

```
#define MIB_COUNTERS 1
```

Most ports limit the maximum buffer size that an SNMP packet can occupy. While the best practical size varies depending on your IP stack and media, 484 is a recommended, safe minimum. Ethernet-based or PPP-based systems can usually use 1400.

```
#define SNMPSIZ   1400     /* MAX size of an snmp packet */
```

If you use SNMP traps, you need to include the trap code with `#define`s as follows:

```
#define ENABLE_SNMP_TRAPS 1
#define V1_IP_SNMP_TRAPS 1
```

SNMP v2 traps might also be supported in the future if it becomes necessary to provide support for them. The macro `V2_IP_SNMP_TRAPS` is reserved for this purpose.

The last macro from `snmpport.h` that porting engineers need to be aware of defines the maximum number of trap targets the end user can configure. This macro controls the size of a static table for trap target information.

```
#define MAX_TRAP_TARGETS 3
```

3.3.3 Testing

If you have not yet hooked up the protocol stack and want to unit test the SNMP Agent, there is a simple method you can use:

1. Hardcode the SNMP packet data into a static buffer.
2. Pass a pointer to it to `snmp_agt_parse()`. The `snmp_agt_parse()` function will return an SNMP reply in your output data buffer. A sample of code, Example 3-4 on page 3-14, is provided to demonstrate this. This example is provided for instruction purposes only.

Example 3-4 Parsing an SNMP request

```
/*
* quick.c
*
* Copyright (C) ARM Limited 1999. All rights reserved.
*
* Sample code to demonstrate a simple way to call the SNMP Agent core to parse
* an SNMP request.
*
*/

/* get system.sysDescr.0 (.1.3.6.1.2.1.1.1.0), using community 'public' */
unsigned char pkt[] = {
    0x30, 0x82, 0x00, ox2D, 0x02, 0x01, 0x00, 0x04, 0x06, 0x70,
    0x75, 0x62, 0x6c, 0x69, 0x63, 0xa0, 0x82, 0x00, 0x1e, 0x02,
    0x02, 0x72, 0x7b, 0x02, 0x01, 0x00, 0x02, 0x01, 0x00, 0x30,
    0x82, 0x00, 0x10, 0x30, 0x82, 0x00, 0x0c, 0x06, 0x08, 0x2b,
    0x06, 0x01, 0x02, 0x01, 0x01, 0x01, 0x00, 0x05, 0x00
};

#define SNMPSIZ   484
unsigned char snmpreply[SNMPSIZ];

void
testAgentParsing()
{
    int reply_len;

    dtrap();            // hook debugger before call...

    reply_len = snmp_agt_parse( pkt, sizeof(pkt), snmpreply, SNMPSIZ );

    dtrap();            // ...and once again afterwards
}
```


3.4 GET operations and scalar variables

The `findVar()` functions referred to in the `variables[]` array (see *Variables structure* on page 2-7), are stubbed out in the MIB Compiler's `.c` output file, and prototyped in the `.h` header file. The stub for the System group, `.iso.org.dod.internet.mgmt.mib-2.system`, is reproduced in Example 3-5:

Example 3-5

```
u_char *
var_system(
        struct variable * vp,    /* IN - pointer to variables[] array */
        oid * name,          /* IN/OUT - input name requested; output name found */
        int * length,        /* IN/OUT - length of input & output oids */
        int oper             /* IN - NEXT_OP (=0), GET_OP (=1), or SET_OP (=-1) */
        int * var_len)       /* OUT - length of variable, or 0 if function */
{
        // TODO: Add code here
        return NULL;          // default FAIL return.
}
```

This routine is called by the SNMP Agent code whenever an SNMP request is received with an Object ID that is matched up with the corresponding Object ID (name) of the routine in the variables table.

Note

The stub produced by the compiler does no work and returns only a `NULL`. The meaning of the returned `NULL` varies depending on the setting of `oper`. If `oper` is nonzero, the SNMP datagram is a `SET` or `GET` command, and the returned `NULL` indicates that an exact match for the variable (passed in `name`) was not available. If `oper` is zero, the request was a `GETNEXT` and the returned `NULL` means that no suitable `GETNEXT` Object ID was found by the routine.

If the routine returns a non-`NULL` value, the return is a pointer to the data of the variable. The type of data pointed to is determined by the variable type in `vp->type`. In the case of a non-`NULL` return, the *`name`*, *`length`*, and *`var_len`* variables must be set to convey information about the data returned. The *`name`* is the Object ID of the returned variable. If the SNMP operation was a `SET` or `GET` (*`oper`* was nonzero), this is the same as the *`name`* passed. If the operation was a `GETNEXT`, the porting programmer must update the *`name`* field. In indexed sequences, this can be difficult. For more details, see the At group example in the demonstration program, or see *GETNEXTs and indexes* on page 3-22.

You must modify the *length* variable to reflect the length of the name returned. The *var_len* value must be set to the length, in bytes, of the variable data returned. For 32-bit numeric returns such as INTEGER, COUNT, and GAUGE, this is four. For Octet strings and Object IDs, it is the length of the string.

As an example, the var_ routine for the System group of variables from MIB-II implements the following variables:

SysDescr Read-only text string describing what the equipment is.

SysObjectID
Read-only Object Identifier, using your Private Enterprise Number.

SysUpTime Read-only measure of how long, in hundredths of a second, this system has been running.

SysContact
Read/write text string describing who is responsible for this equipment.

SysName Read/write text string containing the network name of this equipment.

SysLocation
Read/write text string containing a description of where this equipment is located.

SysServices
Read-only bitfield identifying the capabilities of this equipment.

This function declaration remains unchanged from the stub generated by the MIB Compiler.

Example 3-6

```
u_char *
var_system(
       struct variable *vp, /* IN - pointer to variables[] entry */
       oid * name,          /* IN/OUT - input name requested; output name found */
       int * length,        /* IN/OUT - length of input & output oids */
       int   oper,          /* IN - NEXT_OP (=0), GET_OP (=1), or SET_OP (=-1) */
       int * var_len)       /* OUT - length of variable, or 0 if function */
{
```


For a GET or SET operation, *oper* will be nonzero, and there must be an exact match between the variable name in the request and the name entry in the variables[] table. If they do not match, the request is not valid for this group. This is signaled by the var_ routine returning NULL. The SNMP core handles the task of returning a suitable NOSUCHNAME error response. The use of the compare() function, which performs lexicographic comparisons of ObjectIDs, is similar to the C library function strncmp():

```
if ( oper && ( compare(name, *length, vp->name,
                       (int)vp->namelen) != 0 ) )
    return NULL;
```

The var_ routine needs to return the actual name of the variable found so that the SNMP core can use it to create the response message. For a simple group such as the System group, the SNMP core will have called the var_ routine with the correct entry from the variables[] table. So, you need only copy the value from the *vp* variable pointer into the *name* argument, and set the length of *name* with the *length* argument. var_ routines that handle tables of information need to modify this *name* value (see *GETNEXTs and indexes* on page 3-22 for details):

```
MEMCPY( name, vp->name, (int)vp->namelen * sizeof(oid) );
*length = vp->namelen;
```

The default size of the data to be returned or set is a 32-bit **long** value. This is not necessarily the best choice for the System group because most of its variables are strings, but for the majority of groups, the variables will be INTEGER, COUNTER, or GAUGE values, all of which are the size of a **long**. This value is overwritten in the switch below for those situations where the size is not 32 bits:

```
*var_len = sizeof(long);
```

SET operations might require range checking. In the System group, the only modifiable variables are strings, all of which have been statically allocated SYS_STRING_MAX bytes of space. The SNMP core does not try to SET a variable that is read-only. Therefore, the only checking that is required is that the strings passed in the SET request fit in the buffer. The SNMP core performs this check if you fill in the set_parms entries to enable range checking and to set the valid range (SET operations are described in detail in *Custom SET operations* on page 3-26):

```
if ( vp->acl == RWRITE ) /* force string length check on sets */
{
    set_parms.do_range = TRUE;
    set_parms.hi_range = SYS_STRING_MAX;
                            /* longest acceptable string */
    set_parms.lo_range = 0; /* shortest acceptable string */
}
```

In the following example, the System group has been implemented as a scattered collection of discrete variables, instead of using the recommended data structure produced by the MIB Compiler. This means you must handle every variable within the group as a special case, intead of relying on the auto-indexing feature that can be used with the recommended data structure. Each of the special cases is handled through a `switch` statement:

```
switch ( vp->magic )
{
```

For `STRING` values, you must return the number of characters in the string, excluding any terminating null character. For `GET` and `GETNEXT`, this determines the amount of data that is copied from the variable and returned in the response packet:

```
case SYSDESCR: /* read-only string */
    *var_len = strlen((char *)sys_descr);
    return (u_char *)sys_descr;
```

For an Object Identifier, you must return the number of bytes, instead of the number of subidentifiers. You can achieve this by multiplying the number of subidentifiers by the size of a subidentifier:

```
case SYSOBJECTID:
    *var_len = sys_id_len * sizeof(oid);
    return (u_char *)sys_id;
```

Modifiable strings are handled in exactly the same way as read-only strings. For ease of coding, the length of the string is always returned, as for `GET` and `GETNEXT` operations. However, the `var_len` value is ignored for `SET` operations, which make use of any range limits specified in the `set_parms` structure:

```
case SYSCONTACT:     /* read/write string */
    *var_len = strlen((char *)sysContact);
    return (u_char *)sysContact;

case SYSNAME:        /* read/write string */
    *var_len = strlen((char *)sysName);
    return (u_char *)sysName;

case SYSLOCATION:    /* read/write string */
    *var_len = strlen((char *sysLocation;
    return (u_char *)sysLocation;
```

The following error message indicates that you have neglected to handle one of the variables in the `switch` statement:


```
    default:
        SNMPERROR("var_system: Unknown magic number");
    }

    /* Not Reached */
    return NULL;    /* default FAIL return. */
}
```

3.4.1 Suggested data structures

One method of optimizing both speed and size in the SNMP Agent is to use the *suggested structures* for holding data associated with MIB variables in groups and sequences. An example of how a large group can be implemented with minimal code is the implementation of the MIB-II *Internet Control Message Protocol* (ICMP) group from the demonstration program. All the ICMP counters in this group are maintained in the suggested structure produced by the compiler in the `.h` header file. The C routine needs only to use the magic number (also produced by the compiler) to index the ICMP structure as a table, and return the 32-bit quantity at the indicated offset. The code is reproduced in Example 3-7.

Magic numbers for the ICMP group and the suggested structure are produced automatically by the compiler in the `.h` header file, as shown in Example 3-7.

Example 3-7 Tokens for the ICMP group

```
/* tokens for 'icmp' group */
#define ICMPINMSGS              0
#define ICMPINERRORS            ICMPINMSGS+4
#define ICMPINDESTUNREACHS      ICMPINERRORS+4
#define ICMPINTIMEEXCDS         ICMPINDESTUNREACHS+4
#define ICMPINPARMPROBS         ICMPINTIMEEXCDS+4
#define ICMPINSRCQUENCHS        ICMPINPARMPROBS+4
#define ICMPINREDIRECTS         ICMPINSRCQUENCHS+4
#define ICMPINECHOS             ICMPINREDIRECTS+4
#define ICMPINECHOREPS          ICMPINECHOS+4
#define ICMPINTIMESTAMPS        ICMPINECHOREPS+4
#define ICMPINTIMESTAMPREPS     ICMPINTIMESTAMPS+4
#define ICMPINADDRMASKS         ICMPINTIMESTAMPREPS+4
#define ICMPINADDRMASKREPS      ICMPINADDRMASKS+4
#define ICMPOUTMSGS             ICMPINADDRMASKREPS+4
#define ICMPOUTERRORS           ICMPOUTMSGS+4
#define ICMPOUTDESTUNREACHS     ICMPOUTERRORS+4
#define ICMPOUTTIMEEXCDS        ICMPOUTDESTUNREACHS+4
#define ICMPOUTPARMPROBS        ICMPOUTTIMEEXCDS+4
#define ICMPOUTSRCQUENCHS       ICMPOUTPARMPROBS+4
```

```
#define ICMPOUTREDIRECTS      ICMPOUTSRCQUENCHS+4
#define ICMPOUTECHOS          ICMPOUTREDIRECTS+4
#define ICMPOUTECHOREPS       ICMPOUTECHOS+4
#define ICMPOUTTIMESTAMPS     ICMPOUTECHOREPS+4
#define ICMPOUTTIMESTAMPREPS  ICMPOUTTIMESTAMPS+4
#define ICMPOUTADDRMASKS      ICMPOUTTIMESTAMPREPS+4
#define ICMPOUTADDRMASKREPS   ICMPOUTADDRMASKS+4
```

The suggested structure is shown in Example 3-8.

Example 3-8 MIB table for the ICMP group

```
/* MIB table for 'icmp' group */

struct icmp_mib {
    u_long   icmpInMsgs;
    u_long   icmpInErrors;
    u_long   icmpInDestUnreachs;
    u_long   icmpInTimeExcds;
    u_long   icmpInParmProbs;
    u_long   icmpInSrcQuenchs;
    u_long   icmpInRedirects;
    u_long   icmpInEchos;
    u_long   icmpInEchoReps;
    u_long   icmpInTimestamps;
    u_long   icmpInTimestampReps;
    u_long   icmpInAddrMasks;
    u_long   icmpInAddrMaskReps;
    u_long   icmpOutMsgs;
    u_long   icmpOutErrors;
    u_long   icmpOutDestUnreachs;
    u_long   icmpOutTimeExcds;
    u_long   icmpOutParmProbs;
    u_long   icmpOutSrcQuenchs;
    u_long   icmpOutRedirects;
    u_long   icmpOutEchos;
    u_long   icmpOutEchoReps;
    u_long   icmpOutTimestamps;
    u_long   icmpOutTimestampReps;
    u_long   icmpOutAddrMasks;
    u_long   icmpOutAddrMaskReps;
};
```

The `var_icmp()` function is based on a stub produced by the compiler in the `.c` file. It handles all 26 ICMP group variables, and requires less than ten lines of code to be added to the stub, as shown in Example 3-9.

Example 3-9

```
u_char *
var_icmp(
    struct variable *vp, // IN - pointer to variables[] entry
    oid *name,           // IN/OUT - input name requested; output name found
    int *length,         // IN/OUT - length of input & output oids
    intoper,             // IN - TRUE if exact match is required (for GETs)
    int *var_len)        // OUT - length of variable, or 0 if function
{
    u_char * cp;         // return pointer

    if(oper &&           /* GET or SET ObjectIDs must match exactly */
        (compare(name, *length, vp->name, (int)vp->namelen) !=0))
            return NULL; /* return NULL if not exact match */

    /* The next two lines set the return variables. These are
    actually only needed for GETNEXTs - GETs and SETs already have
    an exact match. */
    memcpy(name, vp->name, (int)vp->namelen * sizeof(oid));
    *length = vp->namelen;
    *var_len = sizeof(long);  /* default length */

    cp = (u_char*)&icmp_mib;
    return(cp + vp->magic);
}
```

It is not always practical to rewrite an existing system to use the suggested structures. The IP stack used with the demonstration program was written before the MIB compiler, and therefore the System group example in *GET operations and scalar variables* on page 3-15 ignores the suggested structure. The suggested structure was retrofitted on the ICMP protocol code. Each structure must be considered individually to determine whether it is worth the effort to use the suggested structures.

3.5 GETNEXTs and indexes

After the ported agent can generate a reply to a request, the remainder of the port, and most of the ongoing development, involves implementing the stub routines. Because this section and the next can be considered advanced stub coding, it is recommended you review Chapter 2 *MIB Compiler* (starting with *Output* on page 2-6, where stub routines are first introduced) before reading the remainder of this section.

A MIB *indexed* variable is a variable that is part of a table, as opposed to the simpler *scalar* variables (non-indexed). Scalar variables, such as those described in the examples in Chapter 2 *MIB Compiler*, only occur once in an SNMP Agent. For example, each agent only has one System group, and by extension, one `sysDescr`, one `sysObject`, and one `sysUptime`. Conversely, indexed variables (tables or ASN.1 SEQUENCEs) can occur multiple times. For example, the MIB-II Interfaces table has a complete set of interface variables for each network interface in the machine. A router with four ethernet cards has four `ifIndex`es, four `ifDescr`s, and four `ifPhysAddress`es. MIB-II stipulates that these are indexed by arbitrarily assigned numbers, 1-4 in this case. The four `ifDescr` instances are represented as `ifDescr.1`, `ifDescr.2`, `ifDescr.3`, and `ifDescr.4`. The actual Object IDs are formed by appending the index (in this case, as in most cases, an ASN.1 INTEGER) to the base Object ID.

An ARM stub routine for an indexed group has to do some extra work in addition to the work done for scalar variables. It has to determine if the Object ID passed has a valid index for the request. This can be problematic when the request is a `SET` or `GETNEXT` request. A `SET` operation might have to create a new table entry if the requested index does not already exist. A `GETNEXT` operation has to generate the lexicographically next variable ID, which might or might not be another instance of the same variable. If it is another instance of the same variable, it might not have a numerically adjacent index value. `SET` operations are discussed in more detail in *Custom SET operations* on page 3-26.

Example 3-10 on page 3-23, with accompanying instructions, contains the `var_ifEntry()` variables routine from the demonstration program, from the file `mib2.c`. It is the first routine in the file to handle an indexed set of variables.

Example 3-10

```
u_char *
var_ifEntry(
    struct variable *vp, /* IN - pointer to variables[] entry */
    oid *name,           /* IN/OUT - input name requested; outputname found */
    int *length,         /* IN/OUT - length of input and output oids */
    int oper,             /* IN - NEXT_OP (=0), GET_OP (=1), or SET_OP (=-1) */
    int *var_len)        /* OUT - length of variable, or 0 if function */
{
```

Rather than simply checking for an exact match on GETs and SETs as in the earlier scalar examples, you must check to see if the index passed as the last component of the array pointed to by *name* (the Object ID from the received packet) is a valid index. For GETs or SETs, this means it must exactly match one of your indexes. For GETNEXTs, it might be an exact match, or it might be an Object ID that is lexicographically lower than a valid response. It must have at least a partial match of a valid interface table Object ID, otherwise the SNMP core code would not have called this function with it. Because the SNMP core code does not know anything about your interface hardware, that information must be used here to put together the SNMP reply:

1. You must first declare some scratch local variables. These include an Object ID buffer, newname[], in which you must build trial Object ID names based on your interfaces. These will be passed to the compare() routine (a strcmp()-like routine for Object IDs) to determine if you have a suitable candidate for a reply:

   ```
   unsigned   interface;
   oid        newname[MAX_NAME_LEN];
   IFMIB      ifp;
   int        result;
   ```

2. Copy the Object ID in the variables table into newname[]. *vp->name* is the Object ID in the variables table that is lexicographically just below the Object ID in the received packet:

   ```
   memcpy((char *)newname, (char *)vp->name,
   (int)vp->namelen * sizeof(oid));
   ```

3. For each interface in the machine, build an Object ID in *newname*[] with that interface's index and run it through compare():

   ```
   for(interface = 0; interface < ifNumber; interface++)
   /* find "next" interface */
   {
       newname[10] = (oid)(interface + 1);
       result = compare(name, *length, newname,
       (int)vp->namelen);
   ```

```
    /* if the operation is a SET or GET and the ObjectIDs
    match, */
    /* break out of the loop */

            if(oper && (result == 0))
                break;
```

If the operation is a `GETNEXT` and *`newname`* is lexicographically greater than the received name, you have found the reply to the `GETNEXT`.

By finding the first valid Object ID for an interface which is greater than the Object ID in the Received packet, you have your `GETNEXT` reply:

```
        if(!oper && (result < 0))
            break;
    }

/* If you looped through all the interfaces without a match, */
/* return NULL; the SNMP core will try the var_ routine for */
/* the next group or table */

    if (interface >= ifNumber) return(NULL);

/* The rest of the routine is similar to examples for the */
/* scalar groups */

    memcpy((char *)name, (char *)newname,
    (int)vp->namelen * sizeof(oid));
    *length = vp->namelen;
    *var_len = sizeof(long);    /* default to 32 bit return */

    ifp = nets[interface]->n_mib;

    switch (vp->magic)
    {

/* handle special cases which are not simple 32-bit counters */
/* from the interface's MIB */

    case IFINDEX:   /* you store them 0 thru n-1, snmp wants */
                    /* 1 through n */
            long_return = ifp->ifIndex + 1;
        return (u_char *)&long_return;
    case IFDESCR:
        *var_len = strlen(ifp->ifDescr);
        return (u_char *)ifp->ifDescr;
    case IFPHYSADDRESS:
        *var_len = nets[interface]->n_hal;
```

```
        return (u_char *)ifp->ifPhysAddress;
    case IFSPECIFIC:     /* could be Oid of ethernet MIB */
        *var_len = sizeof(oid00);
        return((u_char*)oid00);
    default:             /* return 32 bit counter from table */
        return (u_char *)(((char *)ifp) + vp->magic);
    }
}
```

Note

Not all indexes are sequentially numbered integers as in Example 3-10 on page 3-23. A table can be indexed by any sequence of numbers. For example, the interfaces in that example can be numbered 4, 58, and 99. In this case, the `for()` loop would have to look these up somehow, perhaps even having to perform a sort first. Alternatively, the `for()` loop has to examine all the entries in the table, keeping a record of the most preferable option. Examples of the latter technique are given in the `mib2.c` file of the demonstration program.

3.6 Custom SET operations

SET operations are similar to GET operations in that they usually require an exact match of the Object ID. However, SETs are more complex because they may need to:

- determine if a particular SET operation is legal
- perform an action such as disabling a network interface
- create new entries in SNMP tables.

You can encounter problems when trying to set an illegal value or type, or trying to set a nonexistent variable when this is not allowed. The SNMP core can detect attempts to set illegal types (for example, trying to set a counter using a string), and generates a BADVALUE error response to the SNMP station that sent the request.

If your var_ routine returns NULL, the SNMP core generates a NOSUCHNAME error response to the SNMP station that sent the request. You should use this mechanism to indicate when an attempt has been made to either set a non-existent variable, or create a new, invalid table entry.

If all that is required for the SET operation is for the data value from the SET request to be written into a memory location, your var_ routine must optionally establish range checking values, and return a pointer to the location to be written. Range checking is enabled using the range fields of the global set_parms structure:

```
struct SetParms {
    …
    int    do_range;
    long   hi_range;
    long   lo_range;
};

extern struct SetParms set_parms;
```

The set_parms structure is cleared to zero before your var_ routine is called.

If your var_ routine sets set_parms.do_range to nonzero, and the value being set is a numeric type, it must have a value between set_parms.lo_range and set_parms.hi_range, inclusive.

If set_parms.do_range is nonzero, and the value being set is a string or Object Identifier, the length of the string or Object ID must be greater than or equal to set_parms.lo_range, and less than or equal to set_parms.hi_range.

If the value to be set is outside the .lo and .hi range, the SNMP core will generate a BADVALUE error response to the SNMP station that sent the request. If set_parms.do_range is zero, no range checking is performed.

You can see an example of range checking with string values in the System group `var_` routine, `var_system()`, in `mib2.c`.

In some cases, you might be required to do more than set an existing variable. For example, you might want to send a message to another task, manipulate some hardware, or create a new entry in a table such as a route table. You can accomplish this by using the other fields of the `set_parms` structure:

```
struct SetParms {
   int (*access_method)( u_char *, u_char, int, u_char *, int );
   struct variable *vp;
   oid             *name;
   …
};
```

If your `var_` routine sets the `access_method` function pointer in the `set_parms` structure, and returns a non-`NULL` value, the SNMP core does not attempt to perform the `SET` operation for you. Instead, it calls the function pointed to by `access_method`. If the `do_range` field is nonzero, the usual range checking is performed before the `access_method` function is called.

The `access_method` function must return one of the following `SNMP_ERR_` values from `snmp.h`:

`SNMP_ERR_NOERROR`
: No error. The `SET` operation succeeded.

`SNMP_ERR_TOOBIG`
: The SNMP packet (either the request or response) is too large.

`SNMP_ERR_NOSUCHNAME`
: The variable in the request does not exist, or cannot be created.

`SNMP_ERR_BADVALUE`
: The value to be set is invalid, or of the wrong type.

`SNMP_ERR_READONLY`
: The specified variable is read-only (see the note following this list).

`SNMP_ERR_GENERR`
: Some unspecified generic error occurred.

Note

If the MIB description of a variable indicates that it is read-only, the SNMP core calls the `var_` routine, but does not call your `access_method` function. You rarely need to generate `SNMP_ERR_READONLY` error responses.

If you are using an `access_method` function, your `var_` routine must still return a pointer that the SNMP core uses to retrieve the set value. This value is obtained by the SNMP core and returned in the response packet to the SNMP station that sent the request.

For a fully annotated example of using the `access_method` function, see *atEntry table, annotated* on page A-2.

Chapter 4
Function Descriptions

This chapter describes the functions and data items used to interface to the SNMP Agent core. It contains the following sections:

4.1 SNMP Agent interface

The functions `snmp_agt_parse()` and `snmp_trap()` comprise the external interface to the portable SNMP Agent. For examples of how to call these functions, refer to either of the following:

`snmpport.c` Uses the ARM low-overhead UDP/IP stack to interface to the SNMP Agent.

`snmpsock.c` Uses traditional Berkeley Sockets to interface to the SNMP Agent.

4.1.1 snmp_agt_parse()

This function is called by the transport stack (or an implementation routine on top of the transport stack) when an SNMP datagram for the SNMP Agent is received. An example of this is a UDP packet with destination port 161. This routine is the sole entry point to the agent from the protocol stack.

Syntax

```
int snmp_agt_parse(u_char *inbuf, unsigned inlength,
                   u_char *outbuf, unsigned outlength)
```

where:

`inbuf` Is a pointer to the beginning of the SNMP datagram.

`inlength` Is the length of *`inbuf`* data.

`outbuf` Is a pointer to a buffer for a possible reply.

`outlength` Is the length of *`outbuf`*.

Return value

Returns one of the following:

0 If there is no reply data in *`outbuf`*.

Number of bytes in the *`outbuf`* snmp reply
If there is reply data in *`outbuf`*.

Usage

The incoming SNMP data is passed in the pointer *inbuf*, with the length of the datagram specified by *inlength*. The *inbuf* pointer must point to the first byte of the ASN.1 data (usually 0x30), and not to a UDP or IP header.

All processing of the SNMP datagram is done during the call to `snmp_agt_parse()`.

The `snmp_agt_parse()` function might leave a reply to the datagram in the buffer indicated by *outbuf*, so it must be big enough to hold any expected reply. According to *RFC 1157*, the minimum size of this buffer must be 484 bytes. For embedded systems whose primary interface is Ethernet or PPP, a size of 1460 is recommended. In any case, the size of this buffer must not be less than the value of the `SNMPSIZ` macro defined in the `snmpport.h` header file.

If `snmp_agt_parse()` returns a nonzero length, the calling code must make sure the reply data in *outbuf* is sent back to the party that sent the SNMP request. On UDP, this involves preserving the incoming port value and IP address.

4.1.2 snmp_trap()

This function takes the passed parameters and builds and sends an SNMP v1 trap. The trap is built in a static buffer provided when `snmp_init()` is called. A trap packet is sent to each of the hosts in the `trap_targets[]` table.

The structure `trapVar` is defined in the header file `snmp_imp.h`. The current version is reproduced here, but you must refer to the file version when writing code.

```
struct trapVar {            /* struct for each trap variable */
    oid      varName[MAX_OID_LEN];/* ObjectId of variable */
    unsigned varNameLen;    /* oid components in varName */
    u_char   varType;       /* ASN.1 type of variable */
    unsigned varValLen;     /* octets in variable data field */
    u_char   *varBuf;       /* the actual variable data */
    unsigned varBufLen;     /* used only by snmp_parse_trap() */
};
```

Syntax

```
int snmp_trap(int trapType, intspecificType,
              int specificVarCount,
              struct trapVar *specificVars)
```

where:

trapType Is one of the predefined SNMP traps, in the range 1-6.

If *trapType* is not 6 (vendor-specific trap), the remaining variables are ignored by SNMP, and may be 0 or NULL.

specificType

Is a vendor-specific type. These are defined by the vendor.

specificVarCount

Is the number of entries in *specificVars*.

specificVars

Is a pointer to an array of `trapVar` structures.

Return value

Returns one of the following:

0 If there is a parse error.

The length of the trap image built in the passed buffer
Otherwise.

Usage

To send a vendor-specific trap with variables attached, you need to allocate (either statically or dynamically) space for an array of these structures, one per variable. You then need to fill in the values for your trap variables prior to calling `snmp_trap()`. After the `snmp_trap()` call returns, the `trapVar` array can be freed or re-used.

4.2 User-required functions

As part of the port work, you must provide the functions described in this section. Most are referenced at other places in the manual, but they are described in detail here. The functions are:

- *SNMPERROR()*
- *send_trap_udp()* on page 4-7
- *GetUptime()* on page 4-8
- *snmp_upc()* on page 4-9
- *snmp_init()* on page 4-10.

4.2.1 SNMPERROR()

This function is called by the SNMP code when it detects an error that is specific to the SNMP protocol. The `SNMPERROR()` function is meant to print messages for the benefit of the programmer during product development. It can be `#ifdef`ed out before shipping, or its output can be directed to an error log or user console. On systems that support `printf()`, `SNMPERROR()` can be `#defined` to `printf()`, as shown in the example below.

Syntax

```
void SNMPERROR(char *msg)
```

where:

msg Is the error message text to print.

Return value

None.

Example

```
#define SNMPERROR(msg)    printf("SNMP ERROR: %s\n", msg);
```

4.2.2 send_trap_udp()

This function is called by the SNMP core code to send the standard traps, such as authentication failure. Because `send_trap_udp()` is called from the SNMP core, you must not change the name and function parameters. This routine simply has to send the trap buffer passed to the IP address specified at UDP port 162. It is usually a very small routine.

Syntax

```
int send_trap_udp(u_char *out_data, int out_len,
                  ip_addr trap_target)
```

where:

out_data Is a pointer to the SNMP trap packet to be sent.

out_len Is the length of the data pointed to by *out_data*.

trap_target
Is a 32-bit IP address of the host to which the trap is sent.

Return value

Returns one of the following:

0 If successful.

–1 If not successful.

The standard SNMP Agent code ignores this return. However, special ports have been implemented which take other action (such as paging an operator) when trap sends fail.

Example

See the code in `snmpport.c` or `snmpsock.c` for examples of how to implement this function.

4.2.3 GetUptime()

This function is called from the SNMP Agent trap generation code to timestamp outgoing traps. Ports which implement MIB-II, including the demonstration program, also use it for the `sysUptime` variable in the system group.

Syntax

```
u_long GetUptime(void)
```

Return value

Returns a 32-bit unsigned value containing the number of SNMP `TIMETICKS` (in .01 second, that is, 100th of a second, intervals) since the system was last rebooted.

Example

```
u_long
GetUptime()
{
      return (100L * cticks / TPS);
}
```

4.2.4 snmp_upc()

This function is only used if you are using the low-overhead UDP interface of the ARM TCP/IP stack to implement your SNMP Agent. If this is the case, you have to define `PREBIND_AGENT` in your `ipport.h` file. This causes the UDP demultiplexor to call `snmp_upc()` with UDP datagrams received on the SNMP port, UDP port 161. This function has to pass these datagrams to `snmp_agt_parse()` so that the SNMP core can parse them and build a response packet, which `snmp_upc()` must then send back to the originating station using `udp_send()`. These responses must be sent to the UDP port on the remote host from which the request was received. This is passed as the argument `port`.

Syntax

```
int snmp_upc(PACKET p, unshort port)
```

where:

p Is a `PACKET` defined by the ARM TCP/IP stack to be a pointer to a network packet structure, with `nb_prot` pointing to the start of the SNMP data, and `nb_len` indicating the number of bytes in the packet.

port Is the UDP port number (in local-byte order) on the remote station from which the packet was received.

Return value

Returns one of the following:

`ENP_NOT_MINE`
: If the SNMP Agent is not active.

0
: If no errors occurred.

other `ENP_` **code**
: If other errors occurred.

Example

An example of `snmp_upc()` for use with the low-overhead UDP interface to ARM TCP/IP is available in the `snmpport.c` file.

4.2.5 snmp_init()

This function is called from the `ip_start()` function of the ARM TCP/IP stack if `INCLUDE_SNMP` has been defined in your `ipport.h` file. If you are not using ARM TCP/IP with the ARM SNMP stack, you need to arrange for this function to be called from your startup code. The `snmp_init()` function must create a UDP network endpoint, such as a socket, and bind it to the SNMP port, UDP port 161. This function must also initialize other resources used by the SNMP stack, such as the trap buffer required for sending SNMP trap messages (see *Trap buffer* on page 4-13). If your system is implementing traps, the `snmp_init()` function must also send an initial `COLDSTART` trap to all of the trap targets.

Syntax

```
int snmp_init(void)
```

Return value

Returns one of the following:

0 If successful.

-1 If not successful.

Example

Sample `snmp_init()` functions are available in the `snmpport.c` and `snmpsock.c` files.

4.3 SNMP port data

When porting ARM SNMP to your own environment, you will need to provide certain data structures in addition to the `variables[]` array discussed in *Variables structure* on page 2-7. You can declare these variables either in the `snmpport.c` file, or elsewhere in your application-specific code. The data structures you need to provide are:

- *SNMP MIB*
- *Communities*
- *System identifier* on page 4-12
- *Trap targets* on page 4-13
- *Trap buffer* on page 4-13.

4.3.1 SNMP MIB

If you are implementing the SNMP group from MIB-II, you need to `#define MIB_COUNTERS` in your `snmpport.h` file and provide this structure. The structure `SnmpMib` is used by the SNMP Agent to track the number and type of requests received, enabling these statistics to be retrieved using an SNMP station. `SnmpMib` is declared in your `snmpport.c` file.

Syntax

```
#ifdef MIB_COUNTERS
struct snmp_mib SnmpMib;
#endif
```

4.3.2 Communities

The SNMP Agent authorizes requests based on the community string passed as part of the request. The table of allowed communities is defined in your `snmpport.c` file, and consists of the actual table of `communityID` structures and a count of the number of table entries used.

Syntax

```
struct communityId communities[] = {
    "public", RONLY,
    "private", RWRITE,
};

int num_communities = (sizeof(communities)/sizeof(struct
communityId));
```

Example

The demonstration program `snmpdemo` shows how these values can be initialized to static default values in `snmpport.c`, and then overridden by values read from a configuration file in the `main()` routine at system startup.

4.3.3 System identifier

The System group of variables requires a variable, `SysObjectID`. This is an object identifier that uniquely describes the vendor's authoritative identification of the network management subsystem contained in the equipment. This value is allocated within the *Structure of Management Information* (SMI) enterprises subtree (`1.3.6.1.4.1`), and provides a simple and unambiguous way to determine the type of equipment being managed. For example, if vendor *Elements, Inc.* is assigned enterprise number 4242, they use the subtree `1.3.6.1.4.1.4242`, and can assign identifiers to different products and product families, for example:

`1.3.6.1.4.1.4242.1`
: For Routers.

`1.3.6.1.4.1.4242.1.1`
: For its Hydrogen Router.

`1.3.6.1.4.1.4242.1.2`
: For its Oxygen Router.

`1.3.6.1.4.1.4242.2`
: For Switches.

`1.3.6.1.4.1.4242.2.1`
: For its Helium Switch.

`1.3.6.1.4.1.4242.3`
: For Hubs.

`1.3.6.1.4.1.4242.3.1`
: For its Nitrogen Hub.

You must obtain a unique Enterprise ID number for your company, and manage your own subtree of identifiers for your products. Currently, enterprise numbers are allocated by *Internet Assigned Numbers Authority* (IANA), who can be contacted at website `http://www.iana.org`.

Syntax

```
oid      sys_id[]   = {1, 3, 6, 1, 4, 1, ENTERPRISE, 1, 1};
unsigned sys_id_len = sizeof(sys_id)/sizeof)oid);
```

where:

ENTERPRISE

Is `#defined` in `snmpport.h` as your enterprise number.

Example

The demonstration program `snmpdemo` uses the ARM enterprise number (`#define ENTERPRISE 4128` in `snmpport.h`) to set the system identifier in `snmpport.c`. You can use the ARM enterprise number while developing your product, but you must obtain your own enterprise number before your product is shipped.

4.3.4 Trap targets

The file `trap_out.c` declares an array of `MAX_TRAP_TARGETS trap_target` structures. This array must be initialized by your startup code in `main()` to contain the IP address and community string to be used for each intended recipient of SNMP trap messages. `MAX_TRAP_TARGETS` is defined in the `snmpport.h` file.

Syntax

```
struct trap_target trap_targets[MAX_TRAP_TARGETS] = {0};
```

Example

The demonstration program `snmpdemo` shows how these values can be initialized to a NULL list in `trap_out.c`, and then overridden by values read from a configuration file in the `main()` routine at system startup.

4.3.5 Trap buffer

If your system is sending SNMP trap messages, it needs some workspace in which to build them. Your `snmp_init()` function in `snmpport.c` must allocate a buffer, either dynamically or statically, and assign `trap_buffer` to point to it. The variable `trap_buffer_len` must be set to the length of the buffer. The two variables `trap_buffer` and `trap_buffer_len` are declared for you in `trap_out.c`. You simply need to assign the correct values to them.

Syntax

```
unsigned trap_buffer_len = 0;
u_char  * trap_buffer;
```

Example

The `snmp_init()` code in `snmpport.c` and `snmpsock.c` shows one way to initialize these variables.

4.4 ASN.1 parse functions

If you are using an `access_method` function to implement a special SET operation, you will need to use these `asn_parse` functions to determine the value that is to be set. The parse functions are:

- *asn_parse_int()*
- *asn_parse_string()* on page 4-17
- *asn_parse_objid()* on page 4-18
- *asn_parse_null()* on page 4-19.

4.4.1 asn_parse_int()

This function parses a 32-bit integer value from an ASN.1 `INTEGER` type.

Syntax

```
u_char *asn_parse_int(u_char *data, unsigned *datalength,
                      u_char *type, long *intp,
                      unsigned intsize)
```

where:

data Points to the start of the object.

datalength
Points to the number of valid bytes available in *data*. On return, it points to the number of valid bytes left in *data*.

type Points to a location that, on return, is filled with the ASN.1 type of the data.

intp Points to a location that, on return, is filled with the value of the ASN.1 data.

intsize Must, on entry, contain the size, in bytes, of the location pointed to by *intp*. For `asn_parse_int()`, this must be the value `4`.

On entry, *datalength* is input as the number of valid bytes following *data*. On exit, it is returned as the number of valid bytes following the end of this object.

Return value

Returns one of the following:

Pointer to the first byte past the end of this object (start of the next object)
If successful.

NULL On any error.

4.4.2 asn_parse_string()

This function parses a sequence of bytes from an ASN.1 `OCTET STRING` type.

Syntax

```
u_char *asn_parse_string(u_char *data, unsigned *datalength,
u_char *type, u_char *string, unsigned *strlength)
```

where:

data Points to the start of the object.

datalength
Points to the number of valid bytes available in *data*. On return, it points to the number of valid bytes left in *data*.

type Points to a location that, on return, is filled with the ASN.1 type of the data.

string Points to a location that, on return, is filled with the value of the ASN.1 data.

strlength Must, on entry, contain the size, in bytes, of the location pointed to by *string*. On exit, this contains the number of characters placed at location *string*.

On entry, *datalength* is input as the number of valid bytes following *data*. On exit, it is returned as the number of valid bytes following the end of this object.

Note

string will not be null-terminated. That is, there is no \0 character appended to the string.

Return value

Returns one of the following:

Pointer to the first byte past the end of this object (start of the next object)
If successful.

NULL On any error.

4.4.3 asn_parse_objid()

This function parses a sequence of object identifiers from an ASN.1 `OBJECT ID` type.

Syntax

```
u_char *asn_parse_objid(u_char *data, unsigned *datalength
                        u_char *type, oid *objid,
                        unsigned *objidlength)
```

where:

data Points to the start of the object.

datalength
Points to the number of valid bytes available in data. On return, this points to the number of valid bytes left in *data*.

type Points to a location that, on return, is filled with the ASN.1 type of the data.

objid Points to an area of at least `MAX_OID_LEN` bytes that is, on return, filled with the value of the ASN.1 data.

objidlength
Contains, on return, the number of subidentifiers placed in *objid*.

On entry, *datalength* is input as the number of valid bytes following *data*. On exit, it is returned as the number of valid bytes following the end of this object.

Return value

Returns one of the following:

Pointer to the first byte past the end of this object (start of the next object)
If successful.

NULL On any error.

4.4.4 asn_parse_null()

This function parses an ASN.1 NULL object.

Syntax

```
u_char *asn_parse_null(u_char *data,
                       unsigned *datalength, u_char *type)
```

where:

data Points to the start of the object.

datalength
Points to the number of valid bytes available in *data*. On return, this points to the number of valid bytes left in *data*.

type Points to a location which is, on return, filled with the ASN.1 type of the data (ASN_NULL).

On entry, *datalength* is input as the number of valid bytes following *data*. On exit, it is returned as the number of valid bytes following the end of this object.

Return value

Returns one of the following:

Pointer to the first byte past the end of this object (start of the next object)
If successful.

NULL On any error.

Appendix A
SNMP ARP Table Interface

This appendix provides a heavily annotated example of the `atEntry` table, which allows existing entries to be modified, and new entries to be created. The `atEntry` table is a complete example of using an `access_method` function (see *Custom SET operations* on page 3-26). This appendix contains the following section:

- *atEntry table, annotated* on page A-2.

A.1 atEntry table, annotated

Example 1-1 atEntry table

```
/*
 * The atEntry table is the SNMP interface to the IP stack's ARP table. It
 * allows entries to be modified, and also allows new entries to be created.
 * The IP stack's ARP table is not kept in any sorted order, so some work is
 * required to implement GETNEXT correctly. This table has three variables per
 * row, all of which are modifiable:
 *
 * - the interface number
 * - the physical media address
 * - the logical network address.
 *
 * The atEntry table is indexed by the interface index, by a constant 1, and
 * by the network address, which means that an index into this table is
 * actually six subidentifiers long:
 *
 *   <ifIndex>.1.<ipaddr>.<ipaddr>.<ipaddr>.<ipaddr>
 */
int add_atEntry( u_char *, u_char, int, u_char *, int );

/* Use #defines to help with extracting values from the object identifier
*/
#define VAR_AT_IFACE_OID        10
#define VAR_AT_NETTYPE_OID      11
#define VAR_AT_IPADDR_OID       12
#define VAR_AT_OID_LENGTH       16

u_char *
var_atEntry(
    struct variable * vp, /* IN - pointer to variables[] entry */
    oid * name,           /* IN/OUT - input name requested; output name found */
    int * length,         /* IN/OUT - length of input and output oids */
    int   oper,           /* IN - NEXT_OP (=0), GET_OP (=1), or SET_OP (=-1) */
    int * var_len)        /* OUT - length of variable, or 0 if function */
{
/* scratch workspace needed to keep track of the best fit found so far */
oid   lowest[DEF_VARLEN];  /* "best fit" object Id */
oid   current[DEF_VARLEN];
struct arptabent * atp;   /* scratch pointer to table entries */
struct arptabent * lowarp = NULL; /* lowest entry we found */
int i;
u_char * cp;
```

```
/* SET operations are handled separately from GET and GETNEXT operations */
if(oper == SET_OP)
{
   /*
    * The full OID of the requested variable should be:
    *
    * .1.3.6.1.2.1.3.1.1.<var>.<IfIndex>.1.<IpAddr>.<IpAddr>.<IpAdd>.<IpAddr>
    *
    * which is 16 subidentifiers long. If not, you cannot process this request.
    */
   if(*length != VAR_AT_OID_LENGTH) /* MUST have complete index! */
      return NULL;                  /* NOSUCHNAME error reply */

   /*
    * You need to use a helper function, add_atEntry(), to do the work.
    * This requires that you set up the set_parms structure to point to
    * this function, with the requested variable name and the entry from
    * the variables[] table.
    */
   set_parms.access_method = add_atEntry;
   set_parms.vp = vp;
   set_parms.name = name;

   /*
    * When the access_method function add_atEntry() returns, the SNMP
    * core needs to be able to obtain the actual value set. The SNMP
    * core obtains that value from the location pointed to by the return
    * value of this, the var_ routine, using the size set in var_len.
    *
    * At this point, you do not know which ARP table entry you are going
    * to access, so you must return a pointer to a static data area, which
    * the access_method will fill in with the correct value.
    */
   switch(vp->magic)
   {
   case ATIFINDEX:
   case ATNETADDRESS:
      /* interface index and IP address are 32-bit values */
      *var_len = sizeof(long);

      /* this is where the actual value is when */
      /* add_atEntry() returns */
      return (u_char*)&long_return;

   case ATPHYSADDRESS:
      /* the size of a MAC address is 6 bytes */
      *var_len = 6;
```

```
      /* this is where the actual value is when */
      /* add_atEntry() returns */
      return return_buf;
   }
}

/*
 * The rest of the work for a SET operation is handled within the
 * access_method function add_atEntry(). The remainder of the
 * var_atEntry() routine implements GET and GETNEXT operations, and shows
 * how to handle the situation where the underlying data has no natural
 * ordering.
 */

/*
 * Fill in basic name for 'current' from the variable that
 * the SNMP core matched against
 */
MEMCPY((char *)current, (char *)vp->name, VAR_AT_OID_LENGTH *sizeof(oid));

/* scan the ARP table for closest match */
for(atp = arp_table; atp < arp_table + MAXARPS; atp++)
{
   if(!atp->t_pro_addr)  /* ignore this entry if not valid */
      continue;

   /*
    * For each valid entry in the ARP table, generate an OID describing
    * that entry
    */
   current[VAR_AT_NETTYPE_OID] = 1; /* type is IP address */

   /* copy IP address into current. */
   cp = (u_char*)&atp->t_pro_addr;
   for(i = VAR_AT_IPADDR_OID; i < VAR_AT_OID_LENGTH; i++)
      current[i] = *cp++;

   /* set interface index in objId */
   current[VAR_AT_IFACE_OID] = (oid)(GET_NET_NUM(atp->net) + 1L);

   /*
    * A GET operation will require an exact match of the requested
    * variable with the synthesized name. If there is a match, make
    * a copy of the matching name, and keep a pointer to the appropriate
    * ARP table entry.
    */
   if (oper)   /* operation is GET */
```



```
    {
       if (compare(current, VAR_AT_OID_LENGTH, name, *length) == 0)
       {
          MEMCPY((char *)lowest, (char *)current,
                         VAR_AT_OID_LENGTH * sizeof(oid));
          lowarp = atp;
          break;  /* no need to search further */
       }
    }
    /*
     * A GETNEXT operation is looking for an entry that is
     * lexicographically greater than the requested variable. If the
     * synthesized name is greater than the requested name, and less
     * than the current best fit that you have found, then it is a
     * better fit than the current best fit. In this case, make a
     * copy of the name, and keep a pointer to the ARP table entry.
     */
    else  /* caller wants closest match */
    {
       if ((compare(current, VAR_AT_OID_LENGTH, name, *length) > 0) &&
          (!lowarp ||
          (compare(current, VAR_AT_OID_LENGTH, lowest, VAR_AT_OID_LENGTH) < 0)))
       {
          MEMCPY((char *)lowest, (char *)current,
                         VAR_AT_OID_LENGTH * sizeof(oid));
          lowarp = atp;
       }
    }
 } /* end for at_mib loop*/

 /*
  * By this point, one of the following will have been found:
  *
  * - an exact match for a GET operation
  * - the nearest next variable for a GETNEXT operation
  * - nothing at all, indicating that there is no next entry in this table.
  *
  * If there is no next entry, return NULL, so that the SNMP core can call
  * the var_routine for the next variable in the variables[] table.
  */
 if(!lowarp)
    return(NULL);  /* no match */

 /*
  * For a successful GET or GETNEXT operation, return the actual variable
  * instance found in name, and the length of the variable name in length.
  */
```

```
    MEMCPY((char *)name, (char *)lowest, VAR_AT_OID_LENGTH * sizeof(oid));
    *length = VAR_AT_OID_LENGTH;

    /*
     * Finally, return a pointer to the location where the variable's value
     * can be found. Note that long_return is a static data item, and not a
     * local variable.
     */
     switch(vp->magic)
     {
     case ATIFINDEX:
         *var_len = sizeof long_return;
         long_return = (unsigned long)lowest[VAR_AT_IFACE_OID];
         return (u_char *)&long_return;

     case ATPHYSADDRESS:
         *var_len = 6;
         return (u_char *)lowarp->t_phy_addr;

     case ATNETADDRESS:
         *var_len = 4;
         return (u_char *)&lowarp->t_pro_addr;

     default:
         SNMPERROR("var_AtEntry: bad magic number");
     }

     return NULL;
}

/*
 * "access_method" function: Arp table add entry routine.
 * Returns an snmp error from snmp.h (0 == no error)
 *
 * The add_atEntry() function is called by the SNMP core after the
 * var_atEntry() function has returned a non-NULL value (indicating that the
 * variable exists), and after the SNMP core has verified the access
 * permissions are valid, that is, after the SNMP core has verifired that the
 * desired variable is in fact a modifiable one.
 *
 * This routine is called with pointers to the value to set from the SET
 * request packet, and to the location at which to set the variable, as
 * returned by the var_atEntry() function. The variables[] table entry for the
 * variable to be set is passed by way of the set_parms structure
 */
int
add_atEntry(
```

```
    u_char *var_val,      /* pointer to asn1-encoded set value*/
    u_char  var_val_type, /* asn1 type of set value */
    int     var_val_len,  /* length of set value */
    u_char *statP,        /* pointer returned by var_atEntry */
    int    statLen)       /* *var_len from var_atEntry */
{
unsigned iface;         /* interface from ObjectId index */
ip_addr arp_ip;         /* IP address fronm ObjectID index */
struct arptabent *atp;  /* scratch pointer to table entries */
u_char mac_buf[6];      /* temporary storage for MAC address */
unsigned asnbuf_len;    /* for use by asn1 parser */
u_char asn_type;        /* for use by asn1 parser */

/* obtain a pointer to the variable to be set */
struct variable *vp = set_parms.vp;

   /*
    * An IP address and an interface number index the ATTable.
    * Extract these index values from the Object ID from the SET request
    * (stored in set_parms by the var_atEntry() function).
    *
    * The oid2bytes() function will convert a sequence of object
    * identifiers into a sequence of bytes. The following code
    * illustrates how it is used to convert the four OIDs representing
    * the IP address index into an ipaddr variable, arp_ip.  arp_ip will
    * be in network-byte order (big-endian), not local-host byte order.
    */
   oid2bytes((char*)&arp_ip, set_parms.name + VAR_AT_IPADDR_OID, 4);

   /*
    * Extract the interface number from the OID
    */
   iface = (int)*(set_parms.name + VAR_AT_IFACE_OID);

   /*
    * It is recommended that you perform range checking whenever practical.
    * The following code illustrates how you should check that the
    * interface index passed in the request does not exceed the number of
    * interfaces in the system. Within SNMP, the interface index values
    * start at one (not zero), therefore you must also check that you have
    * not received a request to use interface zero:
    */
   if( iface < 1 || iface > ifNumber )
   {
      if(vp->magic == ATIFINDEX)
         return SNMP_ERR_BADVALUE;
```

```
      else
         return SNMP_ERR_NOSUCHNAME;
   }

   /*
    * Use the index values passed to see if there is already an ARP table
    * entry that can be overwritten.
    */
   for(atp = arp_table; atp < arp_table + MAXARPS; atp++)
      if(atp->t_pro_addr == arp_ip)
         break;

   /*
    * If there is no pre-existing entry, you must create one
    */
   if( atp >= &arp_table[MAXARPS] )
   {
      atp = make_arp_entry(arp_ip, nets[iface-1]);
   }

   /*
    * The make_arp_entry() function never fails. It locates an empty
    * ARP table entry and uses that, or it uses the least-recently
    * used table entry.
    *
    * atp will now point to a valid ARP table entry. Use the asn_parse_
    * functions to parse the actual value passed in the SNMP SET request
    * to discover what value to set. The asn_ functions expect to be
    * passed a pointer to the number of bytes left in the buffer to be
    * parsed, and will update that value by subtracting the number of bytes
    * used to parse the current value. This function gets passed the length
    * of the data part of the asn.1 sequence, which excludes the two bytes
    * of header/length information. For the asn_ funtions to be able to
    * work correctly, add two to the passed var_val_len, and use that
    * as the value passed into the asn_ function.
    */

   asnbuf_len = var_val_len + 2;

   /*
    * For each variable that can be set, parse the value passed in the SET
    * request message.
    */

   switch (vp->magic)
   {
   case ATIFINDEX:
```

```
      /*
       * The interface index is passed as an integer. SNMP indexes the
       * interfaces starting at one. The TCP/IP stack indexes them starting
       * at zero, so ensure that you compensate for this.
       */
      asn_parse_int( var_val, &asnbuf_len, &asn_type, (long *)&long_return,
                                    sizeof(long_return) );

      if( long_return < 1 || long_return > (long)ifNumber)
         return SNMP_ERR_BADVALUE;

      atp->net = nets[(int)long_return-1];
      break;

   case ATPHYSADDRESS:
      /*
       * The MAC address is coded as an OCTET STRING, which is parsed into
       * mac_buf.  The system expects a MAC address to be no more than six
       * bytes long, so it is checked here to prevent buffer overflow.
       * The undocumented convention for the AT group is that attempting to
       * set a null MAC address should result in the table entry being
       * deleted. The IP stack represents an unused ARP table entry by
       * the protocol address field being set to zero. Therefore,
       * legitimate lengths for the OCTET STRING are zero and six bytes.
       */
       if( var_val_len && var_val_len != 6)
          return SNMP_ERR_BADVALUE;

       asn_parse_string(var_val, &asnbuf_len, &asn_type, mac_buf,
                                    (unsigned*)&var_val_len);

       if(var_val_len == 0)
          atp->t_pro_addr = 0L;    /* mark it deleted */
       else
          MEMCPY(atp->t_phy_addr, mac_buf, 6 );
       break;

   case ATNETADDRESS:
      /*
       * The Network Address (Protocol Address) is an OCTET STRING of
       * length four.  The asn_parse_ functions do not check that the
       * type they are parsing matches the data that is being parsed,
       * so you can use the asn_parse_int() function to parse the
       * OCTET STRING into an integer variable. The asn_parse_int()
       * function also handles the conversion from network-byte order
       * to local-host byte order. However, the ARP table entry is
       * expected to be stored in network-byte order, so you must
```



```
        * change the value back to network-byte order
        */
        asn_parse_int(var_val, &asnbuf_len, &asn_type, (long *)&long_return,
                                    sizeof(long_return) );
        atp->t_pro_addr = htonl(long_return);
        break;

    default:      /* should not happen */
        dtrap();
    }

    /*
     * Each of the cases above has left the value that was set in the global
     * variable whose address was returned by the var_artEntry() function. This
     * value is obtained by the SNMP core and is used to create the
     * response message that is sent back to the SNMP station that originated the
     * request.
     *
     * The final step is to let the SNMP core know that no errors occured.
     */
    return SNMP_ERR_NOERROR;

    USE_ARG(statLen);
    USE_ARG(statP);
    USE_ARG(var_val_type);
}
```

Appendix B
Building the Demonstration Program

This appendix details the requirements, installation procedure, and steps required to build the demonstration program. Instructions are provided for using both ARM SDT for Windows and ARM SDT for command-line environments.

This chapter contains the following sections:

B.1 About the demonstration program

The demonstration SNMP Agent shipped with ARM SNMP implements all of the following:

- a subset of MIB-II from *RFC 1213*
- a modified version of the *Coffee Pot MIB*, extracted from *RFC 2325*
- an application-specific extension that has been appended to the *RFC 1213* `.mib` file.

The MIB Compiler is used to generate the `.h` header file, the `.c` stubs file, a `.num` numbers file, and the `snmpvars.c` file. However, the demonstration agent only makes use of the `snmpvars.c` file and the `.h` header file.

The `.c` stubs file is replaced by `mib2.c` which contains example implementations of the MIB-II, Coffee Pot MIB, and the application-specific `var_` and `access_method()` routines for the following groups:

- MIB-II System group
- MIB-II Interfaces group
- MIB-II `ifEntry` group
- MIB-II `atEntry` group
- MIB-II `ip` group
- MIB-II `ipAddrEntry` group
- MIB-II `ipRouteEntry` group
- MIB-II `ipNetToMediaEntry` group
- MIB-II `icmp` group
- MIB-II `tcp` group
- MIB-II `tcpConnEntry` group (not implemented)
- MIB-II `udp` group
- MIB-II `udpEntry` group
- MIB-II `egp` group (not implemented)
- MIB-II `egpNeighEntry` group (not implemented)
- MIB-II `snmp` group
- application-specific MIB `snmpDemo` group
- Coffee Pot MIB coffee group
- Coffee Pot MIB `potMonitor` group.

B.2 Requirements

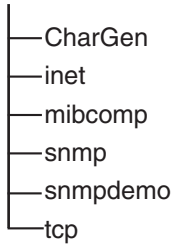
You need the following products to build the demonstration program:

- ARM TCP/IP
- ARM PPP (optional)
- ARM SNMP Agent
- ARM Development Board (PID7T), with Ethernet Kit
- ARM SDT, version 2.50 or later
- Multi-ICE or EmbeddedICE interface.

B.3 Installation procedure

Install the ARM TCP/IP software by following the detailed instructions in *Porting TCP/IP Programmer's Guide* provided with the ARM TCP/IP software. Check that you can compile and run programs on the ARM Development Board (PID7T).

Unpack the ARM SNMP Agent software into the directory containing the ARM TCP/IP sources. You should now have a directory structure similar to:



Note

There might be other directories for PPP, and other ARM networking protocols that you have purchased.

B.4 Building using ARM SDT for Windows

This section details the steps you need to take to build the MIB Compiler and demonstration SNMP Agent using ARM SDT in a Windows environment.

B.4.1 Building the MIB Compiler

To build the MIB Compiler:

1. Using the *ARM Project Manager* (APM), open the `mibcomp.apj` file in the `…\mibcomp` directory.
2. Select **Build mibcomp.apj** from the **Project** menu. The MIB Compiler should build without errors or warnings.

B.4.2 Compiling the example MIB files

To compile the example MIB files:

1. Select **Debug mibcomp.apj** from the **Project** menu. This invokes the *ARM Debugger for Windows* (ADW).

   Note

   You may have to reconfigure the ADW to use the ARMulator, configured for little-endian ARM7 operation, if your ARM Development Board (PID7T) is not configured for little-endian operation with an ARM7 compatible core, or if it has insufficient memory to compile the MIB files.

2. In ADW, select **Set Command Line Arguments** from the **Options** menu.
3. In the command-line arguments box, enter:
   ```
   -i ..\snmpdemo\rfc1213.mib ..\snmpdemo\rfc2325.mib -chnvq
   ```
4. Click **OK**.
5. Select **Go** from the **Execute** menu. You should see output similar to the following:
   ```
   MIB compiler v1.4.2
   Copyright (C) ARM Limited 1999.  All Rights Reserved.
   Copyright 1996 by InterNiche Technologies Inc.
   Copyright 1993 by NetPort Software
   Parsed 450 objects
   ```
6. If you reconfigured the ADW to use the ARMulator, change it back to use the ARM Development Board (PID7T) again.

7. Exit ADW.
8. Move the four output files (`coffee_p.c`, `coffee_p.h`, `coffee_p.num`, and `snmpvars.c`) from the …`\mibcomp` directory to the …`\snmpdemo` directory.

B.4.3 Building the SNMP Agent application

To build the ARM SNMP Agent application:

1. Using the APM, open the `snmpdemo.apj` file in the …`\snmpdemo` directory.
2. Select the appropriate project for your configuration (for example, ARM or Thumb, or big-endian or little-endian).
3. Select **Build…** from the **Project** menu. The SNMP Agent project should build without errors or warnings.

B.5 Building using ARM SDT from the command line

To build the MIB Compiler and demonstration SNMP Agent application using ARM SDT in a command-line environment, you must do the following:

1. Change the directory to …`\snmpdemo`.
2. Edit the makefile.
3. Uncomment the `WHICHVARIANTS=` line, and fill in the variant(s) you would like to build. You can choose from the following:
   - `ArmLittleDebug`
   - `ArmLittleRelease`
   - `ArmBigDebug`
   - `ArmBigRelease`
   - `ThumbLittleDebug`
   - `ThumbLittleRelease`
   - `ThumbBigDebug`
   - `ThumbBigRelease`.
4. Do one of the following:
   - run `make -s all` (UNIX systems)
   - run `armmake -s all` (PC systems).

   The SNMP Agent project should build without errors or warnings.

B.6 Running the SNMP Agent application

To run the SNMP Agent application:

1. Edit the `ether.nv` file and set valid IP addressing options, and SNMP community and trap target information, before running the `snmpdemo.axf` file on the ARM Development Board (PID7T).
2. Use your preferred SNMP network management software to interrogate the SNMP Agent.

Index

The items in this index are listed in alphabetical order, with symbols and numerics appearing at the end. The references given are to page numbers.

N

O

P

R

S

T

U

V

Symbols

