PEER NETWORKS

PATROL® SNMP Toolkit

General Porting Guide

Printing 1.10

October 24, 1996

BMC® SOFTWARE

This Page Left Blank Intentionally

Chapter 1 — Introduction

This *Porting Guide* accompanies the PEER Networks division PATROL® SNMP Toolkit, previously known as the ***OPTIMA*** SNMP Master Toolkit. This printing introduces features new in Release 2.3, and also covers earlier releases. The toolkit helps make applications manageable by supporting rapid development of sub-agent extensions (MIBs) for those applications. The *Porting Guide* covers how to install and build the toolkit for the many platforms already supported, and how to configure the *OptiMaster* Master Agent component. It also leads you through the port to other machines and operating systems. It describes the trace capability.

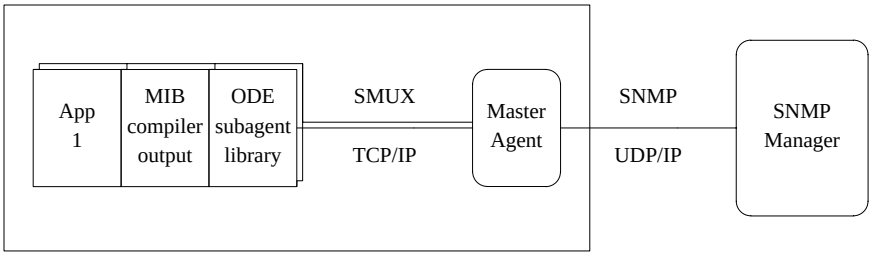
The package includes `INSTALL` scripts, the `Makefile`s and source code for:

- the *OptiMaster* Agent, including its own MIB definitions and a basic configuration file;
- the *OptiGen* MIB Compiler for processing MIB definitions and generating access functions that work with the run-time libraries;
- the *OptiPro* Object Development Environment, including the sub-agent run-time libraries, notably the code that implements the management APIs;
- the *OptiMate* Encapsulator, for incorporating other SNMP agents on the same processor;
- an example of a MIB-II implementation for SunOS; and
- several example manageable applications accompanying the tutorial sections in the *Programmer's Guides*.

This introduction gives an overview of the PATROL SNMP Toolkit management architecture, explains how the components of this package are used, and outlines key porting issues for the components. It also provides a list of relevant RFCs and how to get them, a summary of related documents, an outline of the rest of this document, and typographical conventions this Guide uses.

1.1 An Overview of the PATROL SNMP Toolkit Management Architecture

The *OptiMaster* Master Agent uses SNMP to communicate with Network Management Systems on behalf of your applications. Using the *OptiPro* development environment, you extend your applications to give them sub-agent functionality, making them remotely manageable. The Application Programming Interface (API) and run-time library take care of providing management access to your software's information structures. They are part of the *Object Development Environment* (ODE) that, together with the output of the MIB compiler, turns your application into a sub-agent of the master agent.



1.2 How the Package Components are Used

To make your applications manageable, you must first decide what aspects should be made visible to management. Using the Concise MIB notation, you describe how managers are to manage your application. Next you define the relationship between this abstract description of your application (the MIB definitions) and how your program actually represents the information of interest. The *OptiGen* MIB compiler uses these MIB definitions and extensions to generate access methods and data structures used by the *OptiPro* run-time library. Your application's calls to the high-level API hand off the SNMP management responsibilities to the run-time library and the access methods generated by the MIB compiler. This makes your application manageable by allowing it to act as a sub-agent.

The *OptiMate* Encapsulator is a specialized sub-agent that allows incompatible and non-extensible SNMP agents to be present on the same processor as the *OptiMaster* Master Agent.

The PATROL SNMP Toolkit product includes source code for the MIB Compiler, the Master Agent, the run-time libraries, and the Encapsulator; you may choose to customize it for your target environment, or use it as it is distributed.

1.3 Distributing the Package Components

The MIB compiler is built and run on your development system. The MIB compiler processes your MIB(s) and produces code specific to each sub-agent.

The master agent, run-time libraries and Encapsulator run on your target environment. The run-time libraries need to be linked with the agent, with Encapsulator, and with your sub-agent(s). Since the libraries are reentrant, you may only need to include one copy on any agent/sub-agent machine if that system supports code-sharing.

1.4 Reference Material

Documents defining SNMP and its management information structure are available as Requests for Comments (RFCs). These RFCs and index are particularly helpful:

RFC 1155	Structure and Identification of Management Information for TCP/IP-based Internets
RFC 1157	A Simple Network Management Protocol (SNMP)
RFC 1212	Concise MIB definitions
RFC 1213	Management Information Base for Network Management of TCP/IP-based internets: MIB-II
RFC 1215	A Convention for Defining Traps for use with the SNMP
RFC 1227	SNMP MUX Protocol and MIB
RFC-index	the current index of RFC's

You may order these documents by sending electronic mail to
`<rfc-info@ISI.EDU>`.

The body of the message should read:
```
Retrieve: RFC Doc-ID: RFC####
```

(where #### is the RFC number you are requesting).

For more information on getting RFC's, send email to
`<rfc-info@ISI.EDU>`.

The body of the message should read:
```
help: ways_to_get_rfcs
```

Historical documents defining SNMPv2 and its management information structure are available as Requests for Comments (RFCs). These RFCs are useful background:

RFC 1441	Introduction to version 2 of the Internet-standard Network Management Framework
RFC 1442	Structure of Management Information for version 2 of the Simple Network Management Protocol (SNMPv2)
RFC 1443	Textual Conventions for version 2 of the Simple Network Management Protocol (SNMPv2)
RFC 1444	Conformance Statements for version 2 of the Simple Network Management Protocol (SNMPv2)
RFC 1445	Administrative Model for version 2 of the Simple Network Management Protocol (SNMPv2)
RFC 1446	Security Protocols for version 2 of the Simple Network Management Protocol (SNMPv2)
RFC 1447	Party MIB for version 2 of the Simple Network Management Protocol (SNMPv2)

RFC 1448	Protocol Operations for version 2 of the Simple Network Management Protocol (SNMPv2)
RFC 1449	Transport Mappings for version 2 of the Simple Network Management Protocol (SNMPv2)
RFC 1450	Management Information Base for version 2 of the Simple Network Management Protocol (SNMPv2)
RFC 1451	Manager-to-Manager Management Information Base
RFC 1452	Coexistence between version 1 and version 2 of the Internet-standard Network Management Framework

The current edition of SNMPv2 documentation is contained in the following RFCs:

RFC 1901	Introduction to Community-based SNMPv2
RFC 1902	Structure of Management Information for Version 2 of the Simple Network Management Protocol (SNMPv2)
RFC 1903	Textual Conventions for Version 2 of the Simple Network Management Protocol (SNMPv2)
RFC 1904	Conformance Statements for Version 2 of the Simple Network Management Protocol (SNMPv2)
RFC 1905	Protocol Operations for Version 2 of the Simple Network Management Protocol (SNMPv2)
RFC 1906	Transport Mappings for Version 2 of the Simple Network Management Protocol (SNMPv2)
RFC 1907	Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2)

1.5 How this Guide is Organized

Section 2 describes how to install and administer the *OptiMaster* Agent.

Section 3 explains how to port the development tools.

Section 4 covers porting the Master Agent and the run-time libraries, listing the operating system library functions that are required. It also discusses the system include files and the C compiler.

Section 5 suggests where you are likely to want to customize the source.

Section 6 covers the trace capability, a debugging aid.

1.6 Typographical Conventions

The following typographical conventions are used throughout this document.

- When a term is first defined, it is shown in *italics*.
- When actual "C" code is given, in fragments, programs or synopses, it is shown in `Courier` font.
- In text, names of functions (or variables used in referenced code) are shown in `Courier` font, as in `mgmt_new_instance()`.
- Throughout the document, **bold** is used to call attention to additions, extensions, or differences.

1.7 PEER Networks division Technical Support

Technical Support is available from our California offices on business days from 9:00a.m. to 6:00p.m., Pacific time.

Telephone:	+1 408 556-0720
FAX:	+1 408 556-0735
Electronic mail:	support@peer.com

This Page Left Blank Intentionally

Chapter 2 — Installing The PATROL® SNMP Toolkit

2.1 Reading The Distribution

Reading the tape media requires the following items:

- **Tape Drive** — The 1/4" cartridge tape is written in QIC-11 format.
- **The "tar" Command** — The tape is written in Unix "`tar`" format, with a blocking factor of 96.
- **Disk Space Required (6.6MB)** — The files from the tape will occupy about 5.1 Megabytes of disk space; the other 1.5 Megabytes is roughly what would be required to hold intermediate output files, object files, libraries, and compiler programs produced by the `INSTALL` script and makefiles on SunOS. Depending on your development environment and target architecture, you may require more space for binaries. If you use certain compiler debugging options, such as '`-g`' with the Sun compilers, the space required may be substantially greater. Object files for the six tutorials require .4 Megabytes in addition.

Create a directory (`/usr/peer` is used in all examples) and `tar` off the distribution. Typical `tar` options: "`xpb 96`". It is best if you do **not** run this command as superuser or root.

By request, you may receive a tarred distribution on 3.5 inch floppy diskette. In this case, the diskettes are MS-DOS formatted, and the source code is contained in two compressed tar files, `TAR1.Z` on the first diskette, and `TAR2.Z` on the second diskette. (Consult with your development system's `man` pages or equivalent for the appropriate commands and options.) These files should be copied to your development system using commands for the first diskette similar to the following, assuming you are copying onto `/tmp`:

```
% mount /floppy
% cp /floppy/TAR1.Z /tmp/TAR1.Z
```

Now copy the file from the second diskette and unmount the floppy:

```
% cp /floppy/TAR2.Z /tmp/TAR2.Z
% umount /floppy
```

The files' contents may be extracted from `/tmp` into the `/usr/peer` installation directory using commands similar to the following:

```
% cd /tmp
% uncompress TAR1.Z
% uncompress TAR2.Z
% cd /usr/peer
% tar -xpf /tmp/TAR1
% tar -xpf /tmp/TAR2
% rm /tmp/TAR1
% rm /tmp/TAR2
```

If the distribution has instead arrived as the file `peer.zip` on a floppy disk, use your system's unzip command to unpack the distribution, being careful to preserve the directory structure with your own user ID. For example, provide the `-d` option with `pkunzip`.

The `tar` or `unzip` command will have created the directory structure:

`agent`	*OptiMaster* master agent
build/src/wrk	
config/src/wrk	
snmpsm/src/wrk	
common	*OptiPro* Run-time libraries
build/src/wrk	
ode/src/wrk	
ports/src/wrk	
smux/src/wrk	
snmp/src/wrk	
demos	sub-agent examples
hello/src/wrk	for Unix platforms
hellowin/src/wrk	for Windows platforms
fthello/src/wrk	for Unix platforms
fixedtbl/src/wrk	for Unix, 32-bit Windows
fixedls/src/wrk	for Unix, Windows
dynamls/src/wrk	for Unix, Windows
tbltrap/src/wrk	for Unix, 32-bit Windows
mib2/src/wrk	for SunOS
syntime/src/wrk	for Unix
gdmoc	*OptiGen* MIB compiler
build/src/wrk	
front/src/wrk	
lib/src/wrk	
snmp_c/src/wrk	
include	Include files for the entire package
ame	
ode	
gdmoc	
subagts	specialized subagents
encaps/src/wrk	*OptiMate* Encapsulator

2.2 Supported Platforms

The toolkit source is extremely portable and already incorporates ports to a wide variety of platforms, through compile-time switches and/or different versions of the compiler, linker or loader scripts. To install the toolkit, run the INSTALL script for your platform as explained below. Then run `make` in the installation directory. To build the tutorial examples, run `make` in their respective directories.

To install the toolkit for 16- and 32-bit Windows systems running Microsoft's Visual C++ SDK Release 1.5.2 or 4.0 on Windows, Windows 95 or Windows NT, run `INSTALL.BAT`, specifying either the `WIN` or `W32` option. No other changes may be

needed. (Since the installation script overwrites compiler scripts in `devtools`, do not rerun INSTALL if you've edited those scripts.) Then run `nmake` with the appropriate makefile in the distribution directory to build the MIB Compiler, the Run-time Libraries the Master Agent and Encapsulator. To create the toolkit's scripts on a Windows platform, execute one of the following commands from the installation directory:

```
INSTALL W32
```

```
INSTALL WIN
```

To install the toolkit for OS/2, run the script `INSTALL.CMD` from the installation directory. No other changes may be needed. (Since the installation script overwrites compiler scripts in `devtools`, do not rerun INSTALL if you've edited those scripts.) Then run the OS/2 makefile in the installation directory to build the MIB Compiler, the Run-time Libraries, the Master Agent and Encapsulator. To create the toolkit's scripts for an OS/2 platform, execute the following command from the installation directory:

```
INSTALL
```

To install the toolkit for the following systems, run the `INSTALL` command, specifying one of the targets listed below for the target platform parameter, or `generic` if the platform is in the list: `acer, aix, att, bsdi, dgux, dynix, hpux, irix, lynx, ncr, psos, qnx, sco, sequent, sgi, solaris, solaris2.5, vrtx, vxworks.` (see the *Release Notes* for the latest list). Then run `make` in the distribution directory to build the toolkit components.

To create the compiler and linker scripts in `devtools`, execute the script from the installation directory as follows:

```
% ./INSTALL [target_platform] [compiler] [linker] [loader]
```

If you do not specify the `target` parameter, the installation script assumes the target platform is `GENERIC`, and will create scripts appropriate for SunOS and other supported platforms. For other commercial platforms not listed, specify `GENERIC` as the target platform; no other changes may be needed.

If you have edited the resulting compiler, linker or loader script(s) in the `devtools` directory, do not rerun `INSTALL`.

To build the MIB Compiler, the Run-time Libraries, the Master Agent and Encapsulator, run `make` in the distribution directory. This runs other makefiles that, in turn, invoke the C compiler, linker and loader scripts that `INSTALL` created in the `devtools` directory. The set of `tool` scripts build the *OptiGen* MIB compiler. The `peer` scripts build the agent components that run on the target environment, that is, the *OptiPro* run-time libraries, the *OptiMaster* master agent and the *OptiMate* encapsulator.

If your development and target systems differ, you may need to edit the respective set of scripts. You may also need to change them to specify local options, to create a toolkit tailored to specific compiler, machine and environmental conditions or, later, to turn on the trace capability. See the file `include/ame/machtype.h` for a list of optional features and capabilities already incorporated in the code and selected at compile time. These are particularly useful if you are porting to an as-yet unsupported platform. The compiler, linker and loader scripts generated by the INSTALL script already incorporates capabilities for the specified platform. For more information on porting to as yet unsupported platforms, see Chapters 3 and 4.

2.3 INSTALL Script Dependencies

The non-Windows `INSTALL` script requires the following:

- Bourne shell (`sh`)
- The following will be invoked by the `INSTALL` script, directly or indirectly. Figure 1 shows the commands that must exist on your development system, i.e., the installation dependencies.

Command	INSTALL Script	Makes	`toolcc/ peercc`	`toold/ peerld`	`toolbld/ peerbld`
`basename`			**x**	**x**	**x**
`cc`			**x**		**x**
`cd`		**x**			
`cp`		**x**			
`echo`	**x**		**x**	**x**	**x**
`grep`	**x**				
`ld`				**x**	
`lex`		**x**			
`make`	**x**	**x**			
`rm`		**x**			
`sh`		**x**			
`tail`		**x**			
`test`	**x**		**x**	**x**	**x**
`yacc`		**x**			

Figure 1 - Install Dependencies

2.4 Master Agent Administration

2.4.1 Starting the Master Agent

The agent program is built in `/usr/peer/agent/build/src/wrk` as `Program.o`. The agent requires two parameters. The first is the name of its configuration file. (See Section 2.4.2, "Agent Configuration File.") The second is the name of its parameter storage file. The agent creates this file if one does not exist, and uses it to simulate non-volatile memory. (See section 5.2, "Non-Volatile Agent Parameters.")

There are several ways to invoke the agent:

- invoke the agent manually as follows. This is most useful during porting and testing, and requires super-user access.

```
% su
# cd /usr/peer/agent/build/src/wrk
# ./Program.o CONFIG NOV&
```

- create a script that includes the following lines:

```
cd /usr/peer/agent/src/wrk
./Program.o CONFIG NOV&
```

and invoke this script during system startup from

`/etc/rc.local` (This process will be your system's SNMP daemon.)

- create a `setuid` version of the agent so that it may be started by non-root users. Of course, only one process can bind to a given UDP port at any point time, so there is little motivation for providing a way of starting multiple agents in a system. Furthermore, this approach presents a potential security hole.

On some platforms, the master agent needs to run with super-user permissions so that it may access privileged ports for SNMP and SMUX. To simplify system administration during development, if your system supports the Network Database Functions (see section 4.2.3), you can avoid running the agent as root or at the standard ports by having your system administrator modify the entries in `/etc/services`. Or on Windows platforms, you can change the entries in `C:/WINDOWS/SERVICES`

from the following:

```
smux      199/tcp   # Access to SMUX Agent
snmp      161/udp   # Access to SNMP Agent
snmp-trap 162/udp   # For receiving SNMP traps
```

to the following, for example:

```
smux      1199/tcp  # DEBUG Access to SMUX Agent
snmp      1161/udp  # DEBUG Access to SNMP Agent
snmp-trap 1162/udp  # DEBUG For SNMP traps
```

Note:

Be sure to have appropriate port settings when you go into production!

Specifying alternate ports in `services` requires that your SNMP manager be configured to use this alternate UDP port (1161 in the example above).

You will find it most convenient to override the agent SNMP listening port by adding a `TRANSPORT` clause to the agent configuration file. The same may be done for SMUX, but see the discussion in the following section. Trap ports can be individually overridden in the MANAGER clause. See the following sections for details on the agent configuration file. (As a last resort, if the system's network services data base functions do not retrieve a port number, alternate ports can be hard-coded in `include/ame/portnums.h`.)

2.4.2 Basic Agent Configuration File

The agent configuration file controls several aspects of agent operation. Those aspects peculiar to SNMPv2 are covered in a later section. The ones with generic applicability are covered here, and fall into the following broad categories:

- initial values for sysContact and sysLocation
- community-based access control
- community-based naming
- trap destinations

- choice of transport protocols
- sub-agent access control
- sub-agent protocol parameters
- relationship of subagents to community-based naming

The remainder of this section, through descriptions of the configuration file grammar and examples, shows how you can use the agent configuration file to meet your needs. The MultiMIB Agent and PATROL® *OptiMaster* releases 1.6 and later do accept the more limited configuration file used by toolkit releases 1.5d and earlier.

For background information, see RFC 1157 for an extensive discussion of how community-based SNMP access control works. A number of concepts from SNMPv2 are used here, even when SNMPv2 protocol is not in use. For background information, see RFC 1445, *Administrative Model for version 2 of the Simple Network Management Protocol (SNMPv2)*

2.4.2.1 Initial Values for sysContact and sysLocation

The `INITIAL` directive allows the initial values for the `sysContact` and `sysLocation` variables from MIB-II to be specified. The grammar for an initialization directive is very simple:

```
INITIAL <varName> <String>

<varName> ::= sysLocation |
              sysContact
```

If the string contains spaces, line breaks, tabs, etc., it must be enclosed in quotes. The value may also be specified in hexadecimal notation. The configuration file processing routines will handle mapping <CR> and <LF> characters into NVT (Network Virtual Terminal) strings. (See RFC 854 for a specification of the NVT conventions.)

Note that these directives have an effect **only** when the agent is started with a non-existent non-volatile memory file. Changing the values in the configuration file and forcing the file to be re-read using a `HUP` signal will **not** cause the values of the MIB variables to change. This is to avoid confusing any managers which may have re-configured those variables.

The following example illustrates the use of the `INITIAL` directive in a configuration file.

```
INITIAL sysLocation "Computer Room A
1190 Saratoga Avenue
Suite 130
San Jose, CA 95129-3433
USA"

INITIAL sysContact "Dorothy Sun
Email: <dorothy@peer.com>
Voice: +1 408 556-0720
Fax: +1 408 556-0735"
```

Of course, the actual values you choose will be specific to each system. If no `INITIAL` directive is present for a variable, the default value chose by the agent is the empty string, unless you customize the startup code in module `system.c` in directory `agent/snmpsm/src/wrk`.

2.4.2.2 Specifying Transport Mappings

The master agent may make use of multiple interfaces for SNMP and SMUX traffic. For each interface which the master agent should use for listening for SNMP packets or SMUX connections, there must be one `TRANSPORT` definition in the configuration file. If no transport definitions are present, the defaults (see above) are used. If *any* SMUX transport mappings are present in the configuration, the default is not assumed. If *any* SNMP transport mappings are present in the configuration, the default is not assumed.

Each transport mapping describes an interface over which the master agent listens for SMUX connections from sub-agents or SNMP requests from managers. Note that sub-agents do **not** derive SMUX transport mappings from the agent configuration file. When a sub-agent calls mgmt_init_env() to initalize its management environment, if the first parameter is NULL, the SMUX port number to send to is derived from the TCP/IP stack's `/etc/services` file, or equivalent; if no SMUX port number is specified in this file, the port number specified in the file `include/ame/portnums.h` is used.

The syntax for the `TRANSPORT` clause is shown below. When `appl_protocol` is `SNMP`, `transport_protocol` defaults to UDP. If `appl_protocol` is `SMUX`, `transport_protocol` defaults to TCP. The IP address in f`addr` defaults to that of the master agent's host. The IP port number defaults as described above.

The syntax for the TRANSPORT clause in the agent configuration file looks like this:

```
TRANSPORT  <name>     SNMP
                [OVER UDP SOCKET]
                [AT <addr>]

TRANSPORT  <name>     SMUX
                [OVER TCP SOCKET]
                [AT <addr>]

addr ::=        <ip-kind> | <rfc1449addr> | <full-ip>
ip-kind ::=           <hostid> |
                <hostid> <portid> |
                <portid>

hostid ::=      <hostname> | <ip>

portid ::=      [PORT | : ] <#>

full-ip ::=           <ip>:<#>
ip ::=                <#>.<#>.<#>.<#>

rfc1449addr ::=       <ip>/<#>
```

Note that there are many different ways of entering an IP address.

The following example configures a master agent to accept SMUX connections at both the standard port and at a non-standard port. It also configures the agent to accept SNMP traffic only at a non-standard port.

```
TRANSPORT  extraordinary   SNMP
           OVER UDP SOCKET
           AT PORT 11161

TRANSPORT  ordinary  SMUX
           OVER TCP SOCKET
           AT PORT 199

TRANSPORT  special         SMUX
           OVER TCP SOCKET
           AT PORT 11199
```

The maximum number of concurrent SNMP and SMUX transports will be limited by your target system's limits on the number of open sockets or file descriptors per process.

2.4.2.3 Using Community Strings

Community strings serve many functions in SNMP. Their most common use is to provide a weak form of authentication. In this capacity, they are part of the access control framework for SNMP. They are also used to help name pieces of information in the network. The following example of how communities can be used may be helpful:

- at IP address 192.146.153.65 at port 161, using community string "public", the value of `{sysContact 0}` is "Fred", and only read access is permitted.
- at IP address 192.146.153.65 at port 161, using community string "RenMinRiBao", the value of `{sysContact 0}` is "Fred", but both read and write access are permitted.
- at IP address 192.146.153.65 at port 161, using community string "subsystem1", the value of `{sysContact 0}` is "Joe", and only read access is permitted.

In this example, two of the community strings provide access to the same information, but with different access control policies. This third community string provides access to another set of information. Note that the manager requires external knowledge to determine whether the first two refer to the same information or to two different pieces of information that happen to have the same value.

A community string may be thought of as a key into a database table with the following information:

- **Spatial Semantics:** *which* set of information (where a system may have multiple instances of the same information classes)
- **Temporal Semantics:** *when* was the information retrieved or when will changes take effect (there may be caching, changes may require a reboot to take effect, etc.)
- **Access Control Policy:** *who* is allowed to use this information? What operations are allowed?

By now it should be clear that the community string is seriously overloaded. To support all the semantics carried by community strings, there are several configuration file entries.

`COMMUNITY` entries determine which community strings are valid. If no `COMMUNITY` entries are present, ANY community will be accepted by the agent. For each community, the set of supported operations is specified by the `ALLOW` clause. The `MEMBERS` clause in a community entry restricts the set of managers allowed to use that community. If the

MEMBERS clause is absent from a COMMUNITY entry, any manager may use that community.

```
<communityDefinition> ::= COMMUNITY <communityName>
                          ALLOW <op> [,<op>]* [OPERATIONS]
                          [AS ENTITY <entityName>]
                          [MEMBERS <addr> [,<addr>] ]

<communityName> ::= <name>

<op> ::= ALL | GET | SET | TRAP |
         RESPONSE | GET-NEXT | GET-BULK |
         INFORM | NOTIFY
```

The keyword ALL serves as a shorthand for listing the possibilities.

The optional AS ENTITY clause provides any naming semantics associated with this community. An entity definition associates spatial/temporal properties with a label. The spatial/temporal entity is defined using the following grammar:

```
<entityDefinition> ::= ENTITY <EntityName>
                       DESCRIPTION <String>
                       [WITH TIME DOMAIN <oid>]

<entityName> ::= <name>
```

The following example shows how a system might be configured to provide read-only and read-write access both to its standards resources as well as a special subsystem.

```
ENTITY  SpecialSubsystem
     DESCRIPTION "The ultra-enhanced subsystem"

COMMUNITY  public
           ALLOW GET OPERATIONS

COMMUNITY  public1
           ALLOW GET OPERATIONS
           AS ENTITY SpecialSubsystem

COMMUNITY  "YingXio'ng"
           ALLOW ALL OPERATIONS

COMMUNITY  "Jia-oTo-ng"
           ALLOW ALL OPERATIONS
           AS ENTITY SpecialSubsystem
           MEMBERS   bigboss.megalith.com, 192.146.153.65
```

Thus, to access information from the "ultra-enhanced subsystem", a manager would have to use either community "public1" or "Jia-oTo-ng". Using community "public" or "YingXio'ng" would access the default spatial-temporal entity. Furthermore, use of the community "Jia-oTo-ng" is restricted to operations with a source address from system "bigboss.megalith.com" or 192.146.153.65; operations with any other source address would not be accepted with this community. Note the close relationship between the community-to-entity mapping and the sub-agent-to-entity association described in the section on controlling sub-agent communications, and how to use it to extend SNMP naming by community string.

To avoid unintended access, be sure the ENTITY, COMMUNITY and ALLOW/DENY SUB_AGENT specifications are complete; that is, taken together, the definitions in the configuration file should cover all aspects of the configuration.

2.4.2.4 Controlling Trap Destinations

MANAGER entries in the configuration file define SNMP trap destinations. Note that destinations specified in this manner are independent of destinations specified by means of the SNMPv2 party MIB.

```
<trapDestination> ::= MANAGER <addr>
                  [ON TRANSPORT <transportName>]
                        [SEND [ALL | NO] TRAPS
                         [TO PORT <#> ]
                         [WITH COMMUNITY <String>]]
```

See above for a description of the meaning of `transportName`. If it is not supplied, whatever is being used for primary SNMP traffic will be used.

If the PORT clause is not supplied, `snmp-trap` from your systems `/etc/services` or whatever you have built into the agent by editing `include/ame/portnums.h` will be used.

If the COMMUNITY clause is absent, the community string "public" will appear in the traps that are sent.

The following example requests that all traps be sent to manager "fred" at both port 162 and 11162, from the non-standard snmp port defined above, using a community string of "deep dark secret".

```
MANAGER   fred ON TRANSPORT extraordinary
     SEND ALL TRAPS TO PORT 162
          WITH COMMUNITY "deep dark secret"
     SEND ALL TRAPS TO PORT 11162
          WITH COMMUNITY "deep dark secret"
```

The number of trap destinations is limited only by available memory.

2.4.2.5 Controlling Sub-Agent Communications

There are two major aspects of sub-agent communications that may be controlled through the configuration file. The first is sub-agent access control and naming, and the second is sub-agent protocol parameters.

The grammar for the control of subagent access and associating subagents to specific entities is fairly complex, owing to the wide range of possible configurations that might be needed. There are two types of subagent access control statements: those explicitly permitting access, and those denying access.

```
<subagentControl> ::= <allowSpec> | <denySpec>

<allowSpec> ::= ALLOW <subagentId> [ON <hostSpec>]
                WITH <passwordSpec> [<entitySpec>]
                [<timeout>]

<denySpec> ::= DENY <subagentId> ON <hostSpec>
               WITH <passwordSpec>
```

There are a number of support productions required to complete these subagent access control descriptions.

```
<subagentId>  ::=  SMUX SUBAGENT <oid> |
              UNSPECIFIED SMUX SUBAGENT[S]

<hostSpec> ::= HOST <hostid> |
             UNSPECIFIED HOST[S]

<passwordSpec> ::= PASSWORD <string>
              UNSPECIFIED PASSWORD[S]

<entitySpec> ::= AS ENTITY <entityName>

<timeout> ::= USING <specificTimeout> TIMEOUT
<specificTimeout> ::= <number> SECOND[S] |
                  NO
```

The relationship between subagents and spatial/temporal entities is closely connected to the extended community string semantics described above. Since an entity represents an application's implementation of a set of managed functionality, the master agent is the coordination point for management of multiple entities through their respective sub-agents. The ALLOW/DENY SUBAGENT clauses specify to the master agent which sub-agents will be managing which entities. Combined with community-to-entity mappings defined earlier, this capability allows multiple applications to take responsibility for different instances of the same MIB group, according to the community string of an incoming SNMP request for an otherwise identical MIB variable.

The `USING ... TIMEOUT` clause allows the system to be configured to robustly handle the case where a sub-agent becomes non-responsive. If a subagent does not produce a response to a request within the specified timeout, the agent terminates the connection. If no timeout value is specified for a given subagent, then **no** timeout is used, i.e., operations may take an arbitrarily long time. It is generally a good idea to set up the configuration file to ensure that a timeout value is applied to all subagents.

Although not enforced by the grammar, the following ordering in the configuration file appears to be the most natural for both reading and writing:

1) specific subagents
    a) on specific hosts
        i) with specific passwords
        ii) with unspecified passwords
    b) on unspecified hosts
        i) with specific passwords
        ii) with unspecified passwords
2) unspecified subagents
    a) on specific hosts
        i) with specific passwords
        ii) with unspecified passwords
    b) on unspecified hosts

i) with specific passwords

ii) with unspecified passwords

The following examples should make all of this much clearer. The first example requires that all subagents connect from `localhost`, and sets a 2-second timeout for all of them.

```
ALLOW UNSPECIFIED SMUX SUBAGENTS ON HOST localhost
      WITH UNSPECIFIED PASSWORDS
      USING 2 SECOND TIMEOUT

DENY  UNSPECIFIED SMUX SUBAGENTS ON UNSPECIFIED HOSTS
      WITH UNSPECIFIED PASSWORDS
```

We can elaborate on the previous example by identifying a specific subagent as responsible for the information belonging to a specific spatial/temporal entity. Furthermore, we can impose a different responsiveness requirement on that subagent.

```
ALLOW UNSPECIFIED SMUX SUBAGENTS ON HOST localhost
      WITH UNSPECIFIED PASSWORDS
      USING 2 SECOND TIMEOUT

ALLOW SMUX SUBAGENT 1.3.6.1.4.1.442.2.1 ON HOST localhost
      WITH PASSWORD 0x0011223344556677
      AS ENTITY SpecialSubsystem
      USING 42 SECOND TIMEOUT

DENY  UNSPECIFIED SMUX SUBAGENTS ON UNSPECIFIED HOSTS
      WITH UNSPECIFIED PASSWORDS
```

The example above says that when the hello-world subagent connects from `localhost` using a password of `0x0011223344556677`, that it will be considered to belong to the spatial and temporal domain of `SpecialSubsystem` (defined above), and that it will be allowed to take up to 42 seconds to respond to requests before being disconnected.

NOTE: Depending on the TCP/IP stack, when a sub-agent is on the same processor as the master agent, the <hostSpec> parameter may need to be specificied as <hostname> or as <ip> or as "127.0.0.1" or as "localhost" to allow proper operation. Operation may also be affected by how the stack recognizes/records the master agent's hostname specified on the sub-agent's call to `mgmt_init_env()`.

2.4.3 SNMPv2-Specific Agent Configuration File Options

This section describes agent configuration file options that are specific to SNMPv2. To create a bilingual SNMP/SNMPv2 master agent, define the manifest `SNMPV2_SUPPORTED` in the script `devtools/peercc` by adding:

```
-DSNMPV2_SUPPORTED
```

to the line that invokes the C compiler. If you have not purchased or are not using the *OptiMaster* agent's SNMPv2 capabilities, you do not need the information in this section.

2.4.3.1 Context Definitions

Context defintions map directly to entries in the context table in the party mib. There are two major types: local contexts and proxy contexts. Since there is almost no information in common between these two, the grammar rules are separate.

```
<contextDefinition> ::= <localContext> |
                        <proxyContext>

<localContext> ::= LOCAL CONTEXT <contextName>
                    VIEW <viewName>
                    REFERS TO ENTITY <entityName>
                       AS <oid>

<proxyContext> ::= PROXY CONTEXT <oid>
                   SOURCE PARTY <oid>
                   DESTINATION PARTY <oid>
                   [AND] CONTEXT <oid>
```

The function of a CONTEXT definition in the configuration file is to inform the agent of the relationship between a specific SNMPv2 context and a spatial-temporal entity, as described above. The VIEW clause specified what view constraints are in effect on this context. One would imagine that it would make more sense to associate views with access control list entries, but this is not the way SNMPv2 currently defines it. Of course, this may change in future versions of SNMPv2.

The following example illustrates how a local context may be defined.

```
LOCAL CONTEXT normalContext
      VIEW normalView
      REFERS TO ENTITY SpecialSubsystem
      AS 1.3.6.1.6.3.3.1.4.127.0.0.1.1
```

See the following section for a description of how VIEW information is configured.

2.4.3.2 View Definitions

Each view definition associates a label with a set of included and excluded subtrees.

```
<viewDefinition> ::= VIEW <viewName>
     [[INCLUDE | EXCLUDE] SUBTREE <oid> [MASK <bitmask>]]+
```

These correspond to enties in the SNMPv2 view table. The bitmask identifies which positions in the specified object identifier are to be regarded as "wildcards" for purposes of pattern matching.

The following example shows how a view might be defined. This view only includes the system and snmp groups from MIB-II.

```
VIEW normalView
     INCLUDE SUBTREE 1.3.6.1.2.1.1
        INCLUDE SUBTREE 1.3.6.1.2.1.11
```

Using the MASK clause permits fine-grained access control. However, fine-grained access control may inflict a performance penalty on processing GET-NEXT and GET-BULK protocol operations.

2.4.3.3 Access Control Lists

Each access control list definition maps directly to an entry in the access control list table defined in the SNMPv2 party MIB.

```
<aclDefinition> ::= ALLOW <op> [,<op>]* [OPERATIONS]
                    SOURCE PARTY <partyName>
                    DESTINATION PARTY <partyName>
                    CONTEXT <contextName>
```

An access control list entry definition controls what operations are permitted for a specific conbinarion of source party, destination party, and context. See above (in the discussion of COMMUNITY configuration) for a description of the possible operations.

The following example shows how two access control list entries might be constructed, using the context defined above and the parties defined in subsequent sections.

```
ALLOWSIMPLE-GET, GET-NEXT, GET-BULK OPERATIONS
     SOURCE PARTY         localhostMgr
     DESTINATION PARTY    localhostAgent
     CONTEXT              normalContext

ALLOWRESPONSE, NOTIFY OPERATIONS
     SOURCE PARTY         localhostAgent
     DESTINATION PARTY    localhostMgr
     CONTEXT              normalContext
```

The first entry allows requests to be processed, the second one allows responses and notifications to be sent.

2.4.3.4 Party Definitions

Party definition constructs provide the information needed to establish initial party table entries.

```
<partyDefinition> ::= PARTY <partyName> ON TRANSPORT <transportName>
                      AT <addr> <AuthPriv>
                      AS <oid>

<AuthPriv> ::= <noAuth> <noPriv> |
               <md5Auth> <noPriv> |
               <md5Auth> <desPriv>

<noAuth> ::= NO AUTHENTICATION

<noPriv> ::= NO ENCRYPTION

<md5Auth> ::= MD5 AUTHENTICATION <key>

<key> ::= <string> AS KEY

<desPriv> ::= DES ENCRYPTION <key>
```

A party definition is used to establish initial values for both authentication and encryption keys, if used.

The following example complements the preceding examples, configuring two parties for use with MD5 authentication.

```
LOCAL
PARTY localhostAgent ON TRANSPORT snmpUDPDomain
        AT 0x7f00000100a1
     USING MD5 AUTHENTICATION
          WITH 0x74686973746869737468697374686933 AS KEY
     AND NO ENCRYPTION
     AS 1.3.6.1.6.3.3.1.3.127.0.0.1.3

LOCAL
PARTY localhostMgr ON TRANSPORT snmpUDPDomain
        AT 0x7f00000100a1
     USING MD5 AUTHENTICATION
          WITH 0x74686973746869737468697374686933 AS KEY
     AND NO ENCRYPTION
     AS 1.3.6.1.6.3.3.1.3.127.0.0.1.4
```

2.4.4 Reloading the Configuration File

Under UNIX, the agent may be told to re-read its configuration file by giving the agent process a `HUP` signal. Under Windows, the master agent's window has a menu item to reload the file. This allows a system administrator to change access restrictions without bringing down the agent. Note that any error message go to the agent's `stderr`, which is not necessarily the same as that of the UNIX process issuing the `HUP` signal. This action causes the agent to issue a warm start trap, if traps are enabled in the agent configuration file.

This Page Left Blank Intentionally

Chapter 3 — Porting The Tools

3.1 toolcc, toolld, and toolbld Commands

The `toolcc`, `toolld` and `toolbld` scripts in `/usr/peer/devtools` are provided to simplify cross-development for building programs, such as the MIB compiler, that are to run on your development system. Any particular compiler options or special invocations should go there. In particular, you may have to modify `toolcc` to invoke your C compiler or cross-compiler correctly. Update `toolld` to reflect any system-specific linker requirements for the partially-linked directories that will be used to build the MIB compiler. Updated `toolbld` to reflect system-specific linker requirements to produce executable programs from partially-linked relocatable ones.

3.2 peercc, peerld, and peerbld Commands

The `peercc`, `peerld` and `peerbld` scripts in `/usr/peer/devtools` are provided to simplify cross-development for building the master agent and run-time libraries, along with other software that will run on your target system. Any particular compiler options or special invocations should go there. You may have to modify `peercc` to invoke your C compiler or cross-compiler correctly. Update `peerld` to reflect any system-specific linker requirements. The `peerbld` script is used to generate executable programs from partially-linked components. Depending on your target system, it may produce a fully-linked executable or yet another partially linked program, with externals to be satisfied at load time.

3.3 MIB Compiler

The *OptiGen* MIB compiler (called `gdmoc`) runs on your development system, and would not typically be ported to your target system. The MIB compiler generates standard C language files. Both the MIB compiler input files (ending with `.smi`) and output files (ending with `.h`) for the *OptiMaster* SNMP Agent and sample programs are included on the distribution tape. This makes it possible to do agent and run-time library port work in parallel with MIB compiler porting work.

3.3.1 Directory Structure

All directories in the software distribution are laid out to facilitate the use of revision control systems such as `RCS` or `SCCS`. Using a revision control system during the porting process will dramatically simplify the installation of code upgrades and bug fixes.

The MIB compiler comprises a "frontend" which parses the input and builds internal data structures, and a "backend" that uses these structures to generate output. The directory layout of gdmoc is as follows:

- The MIB compiler frontend and backend are combined into a single executable in:

  ```
  /usr/peer/gdmoc/build/src/wrk
  ```

- The compiler frontend that parses the input is in:

  ```
  /usr/peer/gdmoc/front/src/wrk
  ```

- The library of routines used by the backend is in:

  ```
  /usr/peer/gdmoc/lib/src/wrk
  ```

- The backend that generates the C language representation of SNMP MIB information is in:

  ```
  /usr/peer/gdmoc/snmp_c/src/wrk
  ```

3.3.2 The *OptiGen* MIB Compiler and the C Preprocessor

To facilitate porting the MIB compiler to non-Unix development systems, the compiler "frontend" uses its own pre-processor, `includer.c` in `/usr/peer/gdmoc/front/src/wrk`, instead of your development compiler's C pre-processor. This works well in Unix environments too. `#include` file processing is supported, along with simple `#define` and `#ifdef` constructs. In general, this will not interfere with the use of macros in C access methods, since the MIB compiler output will still be compiled by your C compiler.

However, if you want the MIB compiler to use the C pre-processor instead, remove the `-D` option for `OWN_PREPROCESS` in `devtools/toolcc`. Note, this may cause you to encounter problems with the code that invokes the C pre-processor to handle `#include` and `#define` directives in MIB compiler input files. This code can be found in the "frontend" of the compiler.

This code is in module `frontend.c`, in `run_frontend()`. You may need to modify the invocation of the C pre-processor in the `system()` call to suit your development host and compiler. Many C compilers support a command line option (e.g., `-E` or `-P`) that is equivalent to invoking the preprocessor directly. If your host does not support `system()` or similar fork-like capabilities, the simplest porting alternative is to invoke the C preprocessor and the MIB compiler consecutively in a script, feeding the preprocessor output into the MIB compiler.

The MIB compiler frontend's parsing logic is generated using the `lex` and `yacc` tools. On systems that do not support these tools, one can still port the MIB compiler by making direct use of the `lex` and `yacc` output files that are included on the distribution tape:

```
lexyy.c
ytab.c
ytab.h
```

The only drawback to using these output files directly is that you cannot easily modify the parser to accept new input formats.

These output files are standard C source that can be compiled with your C compiler. In some environments where `stdin` and `stdout` are macros you may need to remove the references to these from file `lexyy.c` The first line of `ytab.c` (as generated by some, but not all versions of `yacc`) may conflict with ANSI and C++ declarations for `malloc()`. The `Makefile` includes an invocation of the `tail` utility to remove the offending line. If you use `bison` instead of `yacc`, the `tail` command should be removed from the `Makefile`.

3.3.3 *OptiGen* MIB Compiler Function Checklist

Figure 2 shows the library function checklist that will help you get the *OptiGen* MIB compiler running on your host platform. If your host environment does not support one of these functions, you will need to implement a work around.

Name	formatting	string	heap	I/O	process
atol	x				
calloc			x		
exit					x
fclose				x	
fflush				x	
fgets				x	
fopen				x	
fprintf	x			x	
fputs				x	
free			x		
fseek				x	
ftell				x	
malloc			x		
perror	x			x	
realloc			x		
sprintf	x	x			
sscanf	x	x			
strcat		x			
strchr		x			
strcmp		x			
strcpy		x			
strlen		x			
strncmp		x			
strrchr		x			
strstr		x			
tmpnam				x	
unlink				x	

Figure 2 - MIB Compiler Library Requirements

Note: Virtually all Unix environments and many Unix work-alikes supply all of these.

If `OWN_PREPROCESS` is **not** defined, the non-POSIX `system()` function is required. The `system()` function may be constructed from `execl()`. Our experience is that `system()` is actually more portable than `execl()`.

Chapter 4 — Porting the Agent and Libraries

This section provides a checklist of library functions required by the *OptiMaster* Agent and *OptiPro* libraries, followed by work arounds for when your system does not support them and finally a section on system include files.

The library dependencies identified here are those that result when the code is built using `-DPEER_GENERIC_PORT` and `-DDES_SUPPORTED` (full SNMPv2 support). Depending on the porting options you select, the library requirements for your target may differ slightly. Dependencies resulting from `lex` and `yacc`-generated code are also omitted, since there is some variation in the code generated by different versions of these tools.

For the identification of system include file dependencies, your compiler probably has an option like the `cc -M` option, to run only the macro preprocessor and generate the `makefile` dependency list. This makes it easier to identify the system-specific dependencies. The `Makefile` dependencies in the code distribution do not include system include files, since their location is highly variable from system to system and they are not prone to frequent change.

4.1 Feature Checklist

Figure 3 is a feature checklist that will help you determine what needs to be done to port the agent and management libraries to your target. It identifies the files that may require modification. In some environments, macros like `FD_ZERO()` may expand into function calls. Second-order effects of this kind are not reflected in this table. The directories most likely to require modification are:

- Agent Directories:
  - `snmpsm /usr/peer/agent/snmpsm/src/wrk`
  - `config /usr/peer/agent/config/src/wrk`
- Library Directories:

— ports /usr/peer/common/ports/src/wrk

— ode /usr/peer/common/ode/src/wrk

Section 4.2 describes what to do for functions not supported by your system.

Features	Agent	Directories	Library	Directories
functions	snmpsm	config	ports	ode
Signals				
sigaction()	main.c			
sigemptyset()	main.c			
Network Database				
gethostbyname()			getownip.c	
gethostname()			getownip.c	
getprotobyname()				opensmux.c
getservbyname()	snmpport.c		opensmux.c	
Socket Library				
accept()	poll.c			
bind()			addrutil.c	
close()			udphooks.c	
			tcphooks.c	
connect()			addrutil.c	
errno			addrutil.c	
			select.c	
listen()				opensmux.c
recv()			tcphooks.c	
recvfrom()			udphooks.c	
select()			select.c	
send()			tcphooks.c	
sendto()			udphooks.c	
setsockopt()			opensmux.c	
socket()	opensnmp.c		opensmux.c	
C printing library				
fprintf()		yyerror.c	maxfiles.c	
fputs()		display.c		
perror()	main.c	frontend.c		opensmux.c
	nonvola.c			poll.c
	opensnmp.c			
C Utility Library				
atol()		parsef.c		
strcat()		parsef.c		
strcmp()		backend.c		
		listutil.c		
		parsef.c		
strcpy()	nonvola.c	parsef.c		
strlen()	nonvola.c		getownip.c	mgmtapi.c
	system.c	parsef.c		
DES Library				
cbc_crypt()	des.c			
heap Library				
calloc()	(numerous)	backend.c	addrutil.c	mgmtapi.c
		frontend.c		reg.c
		listutil.c		smuxsess.c
		parsef.c		
free()	(numerous)	backend.c	addrutil.c	(numerous)
		cleanup.c		
		parsef.c		
malloc()	(numerous)	backend.c		(numerous)
		parsef.c		
realloc()	mgrpart.c			oidtbl.c
	policy.c			
	smuxpart.c			
	view.c			
System calls				
fclose()	nonvola.c	frontend.c		
fopen()	nonvola.c	frontend.c		
fread()	nonvola.c			

Features	Agent	Directories	Library	Directories
functions	snmpsm	config	ports	ode
fwrite() gettimeofday() sysconf()	nonvola.c	getnow.c	maxfiles.c	

Figure 3 - Agent/Library Functions

NOTE: In /usr/peer/common, the directories smux/src/wrk and snmp/src/wrk have no other dependencies.

4.2 My System Doesn't...

The following is an elaboration on what to do if your target environment doesn't support the library functions listed in figure 3 above.

4.2.1 Signals

Signals are used in only one module, main.c in the agent/snmpsm directory. The existing code is necessary for correct operation under Unix, and provides two capabilities:

1. After an SMUX client has been killed or terminated, the agent may get a SIGPIPE signal when it tries to communicate with that client. The program must catch that signal if it is to keep running.

2. The SIGHUP signal is used to tell the agent to re-read its configuration file.

If the target doesn't support signals, be sure to test scenario #1 to make sure the client's demise is detected properly.

Note: version 1.4 used BSD-style signal() to accomplish this. Version 1.5 uses the POSIX-style sigaction() mechanism. If your system does not support the POSIX-style calls, be sure to check whether its signal behavior is BSD or AT&T-style. (In the latter case, you'll need to call signal() again within the signal handler.)

4.2.2 DES Library

The DES (Data Encryption Standard) implementaton is invoked through cbc_crypt(), if you have the have SNMPv2 support, have enabled this functionality by including -DDES_SUPPORTED in the compiler invocation, and have the domestic version of the product. US Export restrictions limit this capability to the domestic version. In the export version, even the references to this function are omitted. Implementations of this function are widely available, both in the US and elsewhere.

4.2.3 Network Database Functions

The common Network Database Functions getprotobyname(), getservbyname(), gethostbyname(), getdomainname(), and gethostname() provide five capabilities:

1. Map symbolic host names from the configuration file into IP addresses.

2. Determine the agent's own IP address. Performance of repeated calls to the internal library function get_own_ip() can be improved by defining the symbol PEER_GETOWNIP_HAS_MEMORY.

3. Determine which ports and protocols to use for SMUX.

4. Determine which port to use for SNMP requests and TRAPS.

5. Determine the domain name for this system to use in the sysName variable in the system group from MIB-II. For most targets the use of `getdomainname()` is disabled, since the NIS domain is not necessarily the same as the fully-qualified host name. The preprocessor macro `PEER_GETDOMAINNAME_MISSING` controls the use of this function.

If these functions or their equivalents are not available, `opensmux.c` in `ode` and `snmpport.c` in `snmpsm` should be modified to always use the default addresses defined in RFC 1157 and RFC 1227 (port number 161 for SNMP and port number 199 for SMUX). The include file `/usr/peer/include/ame/netdbs.h` should be modified appropriately.

The determination of the agent's own IP address is critical; it is a mandatory field in TRAPs. If hostname lookup is not supported, the configuration file grammar should be modified to allow only numeric host addresses.

The determination of the system name can easily be replaced by other local mechanisms in non-Unix environments. Whether a remote manager should be allowed to change a system's domain and name must be considered in light of a network's security policy.

4.2.4 Socket Library

The mechanism used to carry the SNMP protocol is UDP/IP. TCP/IP carries the SMUX sessions between the agent and SMUX clients (sub-agents). If your environment does not support these protocols or the socket API, you will have to find a substitute.

It is important to keep in mind that the services required by the agent are very simple: the SNMP side of things only needs to send and receive datagrams, while the SMUX side only requires a reliable byte stream. Almost any IPC or RPC mechanism can meet this requirement.

Many embedded systems favor an event-driven processing paradigm. Once past the `select()` calls from `select.c` in `ports` from `poll.c` in `snmpsm` and from `smuxsess.c` in `ode`, the agent and library code are event driven. The use of `select()` is merely a concession to how Unix programs have to deal with multiple input streams. In other environments, other control structures may be simpler to use and provide better performance.

The references to `errno` are macros in some embedded systems, which typically expand to function calls to retrieve the correct information. In a few environments, `errno` handling for sockets is separated from that for other files. The include file `ame/errno.h` handles these conversions for the targets to which the code has already been ported.

There are two calls to `setsockopt()` from `opensmux.c` in `ode`. If your system does not support this function, both may safely be removed. One (with the `SO_REUSEADDR` parameter) is primarily of interest when you're testing an implementation, starting and killing the agent in rapid succession. The other (with the `SO_KEEPALIVE` parameter) of interest when agents and subagents are on different systems and there is not prompt notification of connection failure. This call enables the use of TCP keepalives on connections between agents and subagents, and is controlled by the `SMUX_KEEPALIVES` preprocessor macro.

4.2.5 C Printing Library

This informal grouping includes `perror()`, `sprintf()` and `fprintf()`. These are only used to display error messages during startup or (re)configuration, and are easily replaced by other mechanisms. The `fputs()` function is used for diagnostics and for generating packet trace messages if the agent has been built with packet tracing enabled.

4.2.6 C Utility Library

These functions are provided as part of virtually every C run-time library. In cases where they are not, replacements may be obtained through PEER Networks division Consulting. Functions such as `strlen()`, `strcmp()`, `strcpy()`, and `strcat()` are often written in assembler for performance. The conversion function `atol()` is only needed for configuration file processing. If you do not use configuration files, it is not needed.

4.2.7 Heap Library

Both the agent and the libraries make intensive use of heap memory. The dynamic nature of SMUX connections and manager-agent exchanges precludes static allocation techniques, and the variable lifetimes of operations and dependent SMUX operations precludes stack-based allocation techniques for many structures.

The fundamental routines are `malloc()` and `free()`, `realloc()` treated as a primitive for performance reasons in most implementations. The other routine, `calloc()`, can be constructed from these and utility functions. If your target does not support heap allocation, PEER Networks division Consulting can provide you with a heap management library suitable for use in embedded systems.

4.2.8 System Calls

- `sysconf()` — this function is needed to determine the width of the select mask. In many environments, this may be replaced by a constant. If you don't use `select()`, you may not need it at all. The non-POSIX equivalent service may be obtained using `ulimit()`.

- `gettimeofday()` — This function is used to determine the system's clock time. In SNMP, time is normally expressed in 1/100 seconds. The module `snmpdt.c` in `ode` will need to be updated to reflect whatever your system's timekeeping convention is, as will `/usr/peer/include/ame/time.h`.

- `fopen()`, `fread()`, and `fclose()` These functions are used to access configuration files. If your system does not have some way of getting to non-volatile configuration data, you'll have to do without it.

- `fwrite()` — This function updates non-volatile parameters. You may choose to provide access to nvram or simply disable this feature by editing `ame/machtype.h`.

4.3 System Include Files

System include file dependencies are generally confined to include files rather than C source files. The following may be of concern during a port. For each one, the data structures that need to be defined are listed, followed by a discussion of alternatives.

```
/usr/peer/include
          ame/errno.h
          ame/fd.h
          ame/portnums.h
          ame/syspub.h
```

4.3.1 ame/time.h

```
struct timeval
struct timezone
```

Under SunOS, these are defined in `/usr/include/sys/time.h`. In other systems, look at `/usr/include/time.h` or possibly `/usr/include/sys/select.h` for the definition. If they are not available in any of these, `#define NEED_OWN_TIME` in the file `include/ame/machtype.h`.

These structures are needed by the system call `gettimeofday()`, and are used to convert system time into SNMP ticks. The second item, `struct timezone`, is only needed for `lint`.

4.3.2 ame/appladdr.h

```
struct sockaddr
struct sockaddr_in
```

These are usually well-buried with a tangled nest of dependencies in various versions of Unix. Replace the `#include` references in `/usr/peer/include/ame/appladdr.h` with whatever your target requires. If you are porting to a non-socket environment, these may change substantially.

4.3.3 ame/netdbs.h

These structures support the library routines `getprotobyname()`, `gethostbyname()`, `gethostname()`, and `getservbyname()`. If you do not need run-time determination of SNMP ports and host addresses, you may be able to avoid these.

These are usually well-buried with a tangled nest of dependencies in various versions of Unix. Replace the `#include` references in `/usr/peer/include/ame/netdbs.h` with whatever your target requires.

4.3.4 ame/errno.h

This file provides the definitions of various error codes. It also includes macros to accomodate the handling of `errno` in various embedded systems where socket error handling has been separated from error handling for other files.

4.3.5 ame/fd.h

This file provides the definitions for the structures and macros used in conjunction with `select()`. If you are going to an environment which does not use `select()`, `poll()`, or something similar, this file will be superfluous.

4.3.6 ame/portnums.h

This file defines default SNMP and SMUX port numbers. The `htons()` and `htonl()` macros, normally hidden somewhere in `<netinet/in.h>` or the like, are needed here.

4.3.7 ame/syspub.h

This file hooks into the prototypes for the library functions. The path names used here are the POSIX / ANSI standard ones. Bracketing is included here for mixed C and C++ environments. If you are porting to a "pure" C++ environment, you may need to removed the `EXTERN_BEGIN` and `EXTERN_END` brackets. Alternatively, the definitions of these macros can be changed in "`ame/machtype.h`". Which is more appropriate will depend on how your target's libraries are structured.

4.3.8 sys/param.h

This file contains `MAXHOSTNAMELEN,` the maximum size of hostname, used by the common Network Database Functions.

4.4 Compiler Issues

The source supplied on this distribution ANSI C code it is written so that it can also be compiled using K&R C compilers with common extensions. The code also permits compilation by C++ compilers, such as Gnu C++. In general, the file `/usr/peer/include/ame/machtype.h` allows control of most interesting compiler dependencies. If you are using unusually old C compilers, there are two issues to deal with:

- If your compiler does not support the `void` construct, editing the file `/usr/peer/include/ame/machtype.h` by defining `void` as an `int` will most likely do the job. However, the MIB compiler must be changed manually.

- Some older compilers almost support `void` except they do not handle pointers to `void (void *)`. For this case, modify the file `/usr/peer/include/ame/machtype.h` by adding

```
#define VOID_NOT_SUPPORTED.
```

The C++ support is written for use in mixed C and C++ environments, for linkage with standard C libraries. If you are porting the code to a "pure" C++ environment, some modifications to "`ame/syspub.h`" and related header files may be necessary.

The MIB compiler output supports compilation by several compilers, including K&R, ANSI, and Gnu C++. If your compiler places unusual restrictions on the kind of input it can process, it may be necessary to modify the *OptiGen* MIB compiler. Fortunately, this is rather straightforward. The file `smi_c.c` in the `snmp_c` directory of `gdmoc` controls how its output is formatted.

This Page Left Blank Intentionally

Chapter 5 — Customization

5.1 Agent Configuration File Support

All the support for agent configuration files is in the directory `/usr/peer/agent/config/src/wrk`. This component is one that you may want to customize for your product.

5.1.1 Logical Requirements

Configuration file support requires:

- some kind of non-volatile storage for the configuration file
- a grammar describing the structure of the human-readable configuration file

5.1.2 How it works

The function `main()` in `main.c` in the directory `agent/snmpsm` invokes `parse_config_file()` in `frontend.c` in `agent/config` passing the name of the configuration file and a pointer to the agent context.

After the configuration file has been analyzed, the information is fed into the agent through calls to the functions `new_mgr_partner()` and `new_community()` which are supplied in the directory `agent/snmpsm`. Configuration processing then cleans up allocated memory and returns.

When the agent receives an SNMP request (PDU), it first checks the "Community" list. The PDU is accepted if either:

- there is an entry which matches the PDU's community and does not limit the set of managers authorized to use that community, or
- there is a matching community entry and the PDU's source address is in that entry's list of authorized managers.

5.1.3 How to Modify it

The grammar of configuration files may be modified, extended, or constrained by changing the `yacc` and `lex` grammar files.

5.1.4 How to Replace It

Your replacement library will be invoked through the entry point `parse_config_file()`, and must feed its data back through `new_mgr_partner()` and `new_community()`. Contact PEER Networks division Technical Support for example code that establishes a simple configuration without a file system -- any SNMP request with community string "public" is accepted, and all traps are sent to a single manager host.

5.2 Non-Volatile Agent Parameters

In addition to the configuration file described above, the agent provides support for remotely manageable parameters which should survive a reboot of the agent system. These attributes control whether authentication traps are enabled, and provide the system location, system name, and system contact strings for the MIB-II system group. The manifest "`NON_VOLATILE_AVAILABLE`" in `<ame/machtype.h>` controls whether any of this support code is enabled. The module `nonvola.c` in the `agent/snmpsm` directory supports this feature, and is the best starting point if you need to provide EEPROM or other special support.

The format of the file simply an image of this C data structure, not intended for human consumption nor for transport. Its contents are specific to the system on which the agent is running, and it is created and updated by the agent as needed. The strings stored in this structure are `NULL` terminated.

```
struct non_volatile_image
{
     ubyte version;
     ubyte snmpEnableAuthenTraps;
     ubyte sysContact[256];
     ubyte sysName[256];
     ubyte sysLocation[256];
};
```

This file image is updated whenever a request to modify any of its components is successfully processed.

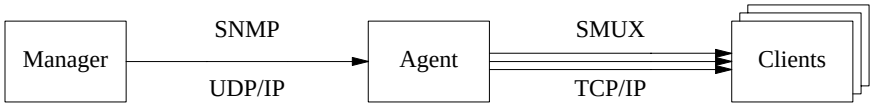
5.3 Constructing Makefiles

It is a good idea to automate `Makefile` generation. Doing so helps ensure that the `Makefile`s correctly reflect dependencies. Most C compilers offer a '`-M`' option to generate dependencies. (In some cases, you might have to invoke the C-preprocessor directly.) The output, with a little `sed` and `awk` massage, produces a good solid `Makefile`.

5.4 Agent/Library Communications

For highly customized ports and ports to non-Unix-like environments, it may be necessary to go deeper into how the *OptiMaster* Agent and libraries actually work. For background, see RFC 1227, *SNMP MUX Protocol and MIB*.

REQUEST



RESPONSE

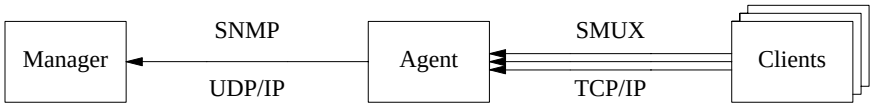


Figure 4 - GET/SET/GET-NEXT model

An SNMP management request ("parent" operation) is delegated by the agent to one or more sub-agents (clients) as SMUX "child" operations. The intermediate results are gathered by the agent, and merged to form the SNMP response to the "parent" operation. In figure 4, a single SNMP "parent" operation causes three SMUX "child" operations to be sent to clients.

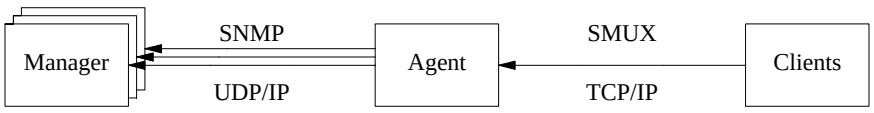


Figure 5 - TRAP Model

When TRAPS are emitted by clients, they are forwarded by the agent to zero or more SNMP managers. The agent will encode each copy of the trap with the correct community string for each manager. This is shown in figure 5.

It should be noted that the agent is also an SMUX client. The agent uses the usual *OptiPro* management API to model the SNMP and system groups from MIB II (RFC 1213) and the SMUX MIB (RFC 1227).

In customizing or porting to other environments, the following options may be of interest:

- use of some transport other than TCP/IP for the SMUX stream.
    - — SMUX requires a reliable, connection-oriented mechanism. many IPC and RPC schemes would work well for this

- — the use of TCP/IP allows configurations where the client is on a different system from the agent
- — changing this requires modifications to `poll.c` in `snmpsm`, to `smuxsess.c` in `ode`, and to `tcphooks.c` in `ports`, among others.

- use of some transport other than UDP/IP for SNMP
  - — this may restrict your degree of interoperability with other products
  - — this requires modifications to `poll.c` in `snmpsm` and to `udphooks.c` in `ports`. You may also need to modify `opensnmp.c` in `snmpsm`.

5.5 Manager Access Control

The set of managers permitted access is controlled by the agent configuration file. It allows access to be restricted by community, by manager address, and by operation. If a new manager is encountered (and passes any Community restrictions) it is added to the working set of partners by a call to `new_mgr_partner()` in `mgrpart.c` from `snmp_msg()` in `snmpsm.c`, both in `agent/snmpsm`. If you want the agent to send TRAPs to dynamically acquired partners, this call should be modified to specify a TRAP address for such partners. Clearly, if you decide to support extensive TRAP filtering capabilities as well, you will have to carefully consider interactions between dynamic partner acquisition and TRAP filtering.

5.6 Client Access Control

The set of clients is determined dynamically as the clients register and deregister with the agent. You may choose to support password-based client authentication as supported by the SMUX protocol, although you should be aware that the protocol, as defined, is no more secure than `rlogin` or `telnet`. Function `smux_open()` in module `smuxsm.c` in `agent/snmpsm` is where such authentication would need to be performed.

Chapter 6 — Trace Capability

To aid in debugging, the toolkit provides a trace capability. Trace levels are set at compile time, by defining one or more manifests. Traces are directed to `stderr`.

The trace control manifests listed later in this section are used in `include/ame/machtype.h`. To ensure code consistency, define the manifests you want in `devtools/peercc`. For example, to trace SNMP packets received and sent, alter the line in `peercc` that invokes the C compiler to read:

```
if ${Compiler} -DPEER_GENERIC_PORT -DSNMP_PACKET_TRACE $@
```

Then rebuild the master agent and/or subagent(s), including the run-time libraries. When rebuilding the code, be sure to link with the module `common/ports/src/wrk/display.c`. (You may tailor this file to fit your system's needs.) Since tracing impacts performance, timeouts are likely to occur. For this reason, only use tracing during development or while doing controlled trouble-shooting or isolation.

The trace control manifests used in `include/ame/machtype.h` are:

```
*
*   Packet Tracing:
*        SMUX_SERVER_TRACE - define if you want the server (agent)
*             to print out information about all SMUX packets
*        SMUX_CLIENT_TRACE - define if you want the client to
*             print out information about all SMUX packets
*        SNMP_PACKET_TRACE - define if you want a trace of all
*             (authenticated) snmp packet activity
*        UDP_TRACE - display all datagram activity
*        UDP_READ_TRACE - display all incoming datagram activity
*        UDP_WRITE_TRACE - display all outgoing datagram activity
*        TCP_TRACE - display all TCP activity
*        TCP_READ_TRACE - display all incoming TCP activity
*        TCP_WRITE_TRACE - display all outgoing TCP activity
*
*   Debug Tracing:
*        PEER_MGMT_API_TRACE - trace invocation of mgmt_* functions
*             in the common/ode library
```

6.1 SNMP protocol tracing

To display in decoded format all SNMP traffic that has passed authentication, use the SNMP_PACKET_TRACE manifest.

6.2 SMUX session tracing

To display in decoded format all SMUX traffic at the master agent, use SMUX_SERVER_TRACE. To display the SMUX traffic for a subagent, use SMUX_CLIENT_TRACE.

6.3 UDP Tracing

The manifest UDP_TRACE causes all UDP data (generally the SNMP exchanges between the master agent and managers) to be displayed. To display only the incoming UDP data, use UDP_READ_TRACE. To display only outgoing UDP data, use UDP_WRITE_TRACE.

6.4 TCP Tracing

The manifest TCP_TRACE causes all TCP data (generally the SMUX traffic between the master and subagents) to be traced. This may be of use at the master agent level, causing all traffic for all subagents to be reported, or at the subagent level, causing just that subagent's dialog to be traced.

To display only the TCP data received by the entity in question, use the TCP_READ_TRACE manifest.

To display only the TCP data transmitted by the entity in question, use the TCP_WRITE_TRACE manifest.

6.5 Management API Invocation Tracing

The PEER_MGMT_API_TRACE manifest enables the tracing of all calls of the *OptiPro* management API functions. It causes the invocation of each of the API functions to be reported, along with the circumstances under which control is returned to the managed application's code. This is most useful for debugging subagents.

Index

D —

E —

F —

G —

H —

I —

K —

L —

M —

N —

O —

P —

Q —

R —

S —

T —

U —

V —

W —

Y —

Z —

Table of Contents

PEER Networks, a division of BMC Software, Inc.

Document No. 20010

This manual is copyrighted by PEER Networks, a division of BMC Software, Inc., with all rights reserved. Under the copyright laws, this manual may not be copied, in whole or in part, without the written consent of PEER Networks, a division of BMC Software, Inc.

This product is the sole and exclusive property of PEER Networks, a division of BMC Software, Inc. and is protected by U.S. and other copyright laws and the laws protecting trade secret and confidential information. This product contains trade secret and confidential information, and its unauthorized disclosure is prohibited. Reproduction, utilization and transfer of rights to the software, whether in source or binary form, is permitted only pursuant to a written agreement.

Unpublished, © PEER Networks, a division of BMC Software, Inc.
All Rights Reserved

RESTRICTED RIGHTS LEGEND

This product is supplied to the U.S. Federal Government as RESTRICTED COMPUTER SOFTWARE as defined in Clause 52.227-19 of the Federal Acquisition Regulations and as COMMERCIAL COMPUTER SOFTWARE as defined in Subpart 227.401 of the Defense Federal Acquisition Regulations. Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software Clause at DFARS 252.227.7013.

PEER Networks, a division of BMC Software, Inc.
1190 Saratoga Avenue, Suite 130
San José, CA 95129-3433 USA
Telephone +1 408 556-0720

Information in this document is subject to change without notice and does not represent a commitment on the part of PEER Networks, a division of BMC Software, Inc.
All trademarks and registered trademarks are properties of their respective owners.