

**NAME**

snmpd.conf - configuration file for the Net-SNMP SNMP agent

**DESCRIPTION**

The Net-SNMP agent uses one or more configuration files to control its operation and the management information provided. These files (**snmpd.conf** and **snmpd.local.conf**) can be located in one of several locations, as described in the *snmp\_config(5)* manual page.

The (perl) application **snmpconf** can be used to generate configuration files for the most common agent requirements. See the *snmpconf(1)* manual page for more information, or try running the command:

```
snmpconf -g basic_setup
```

There are a large number of directives that can be specified, but these mostly fall into four distinct categories:

- those controlling who can access the agent
- those configuring the information that is supplied by the agent
- those controlling active monitoring of the local system
- those concerned with extending the functionality of the agent.

Some directives don't fall naturally into any of these four categories, but this covers the majority of the contents of a typical **snmpd.conf** file. A full list of recognised directives can be obtained by running the command:

```
snmpd -H
```

**AGENT BEHAVIOUR**

Although most configuration directives are concerned with the MIB information supplied by the agent, there are a handful of directives that control the behaviour of *snmpd* considered simply as a daemon providing a network service.

agentaddress [<transport-specifier>:]<transport-address>[,...]

defines a list of listening addresses, on which to receive incoming SNMP requests. See the section **LISTENING ADDRESSES** in the *snmpd(8)* manual page for more information about the format of listening addresses.

The default behaviour is to listen on UDP port 161 on all IPv4 interfaces.

agentgroup {GROUP#GID}

changes to the specified group after opening the listening port(s). This may refer to a group by name (GROUP), or a numeric group ID starting with '#' (#GID).

agentuser {USER#UID}

changes to the specified user after opening the listening port(s). This may refer to a user by name (USER), or a numeric user ID starting with '#' (#UID).

leave\_pidfile yes

instructs the agent to not remove its pid file on shutdown. Equivalent to specifying "-U" on the command line.

maxGetbulkRepeats NUM

Sets the maximum number of responses allowed for a single variable in a getbulk request. Set to 0 to enable the default and set it to -1 to enable unlimited. Because memory is allocated ahead of time, setting this to unlimited is not considered safe if your user population can not be trusted. A repeat number greater than this will be truncated to this value.

This is set by default to -1.

maxGetbulkResponses NUM

Sets the maximum number of responses allowed for a getbulk request. This is set by default to 100. Set to 0 to enable the default and set it to -1 to enable unlimited. Because memory is allocated ahead of time, setting this to unlimited is not considered safe if your user population can not

be trusted.

In general, the total number of responses will not be allowed to exceed the `maxGetbulkResponses` number and the total number returned will be an integer multiple of the number of variables requested times the calculated number of repeats allow to fit below this number.

Also not that processing of `maxGetbulkRepeats` is handled first.

### SNMPv3 Configuration - Real Security

SNMPv3 is added flexible security models to the SNMP packet structure so that multiple security solutions could be used. SNMPv3 was original defined with a "User-based Security Model" (USM) [RFC3414] that required maintaining a SNMP-specific user database. This was later determined to be troublesome to maintain and had some minor security issues. The IETF has since added additional security models to tunnel SNMP over SSH [RFC5592] and DTLS/TLS [RFC-to-be]. Net-SNMP contains robust support for SNMPv3/USM, SNMPv3/TLS, and SNMPv3/DTLS. It contains partial support for SNMPv3/SSH as well but has not been as extensively tested. It also contains code for support for an experimental Kerberos based SNMPv3 that never got standardized.

Hopefully more SNMP software and devices will eventually support SNMP over (D)TLS or SSH, but it is likely that devices with original support for SNMP will only contain support for USM users. If your network manager supports SNMP over (D)TLS or SNMP over SSH we suggest you use one of these mechanisms instead of using USM, but as always with Net-SNMP we give you the options to pick from so you can make the choice that is best for you.

### SNMPv3 generic parameters

These parameters are generic to all the forms of SNMPv3. The SNMPv3 protocol defines "engineIDs" that uniquely identify an agent. The string must be consistent through time and should not change or conflict with another agent's engineID. Ever. Internally, Net-SNMP by default creates a unique engineID that is based off of the current system time and a random number. This should be sufficient for most users unless you're embedding our agent in a device where these numbers won't vary between boxes on the devices initial boot.

EngineIDs are used both as a "context" for selecting information from the device and SNMPv3 with USM uses it to create unique entries for users in its user table.

The Net-SNMP agent offers the following mechanisms for setting the engineID, but again you should only use them if you know what you're doing:

**engineID STRING**

specifies that the engineID should be built from the given text STRING.

**engineIDType 1|2|3**

specifies that the engineID should be built from the IPv4 address (1), IPv6 address (2) or MAC address (3). Note that changing the IP address (or switching the network interface card) may cause problems.

**engineIDNic INTERFACE**

defines which interface to use when determining the MAC address. If *engineIDType 3* is not specified, then this directive has no effect.

The default is to use `eth0`.

### SNMPv3 over TLS

SNMPv3 may be tunneled over TLS and DTLS. TLS runs over TCP and DTLS is the UDP equivalent. Wes Hardaker (the founder of Net-SNMP) performed a study and presented it at an IETF meeting that showed that TCP based protocols are sufficient for stable networks but quickly becomes a problem in unstable networks with even moderate levels of packet loss (~ 20-30%). If you are going to use TLS or DTLS, you should use the one appropriate for your networking environment. You should potentially turn them both on so your management system can access either the UDP or the TCP port as needed.

Many of the configuration tokens described below are prefixed with a '[snmp]' tag. If you place these tokens in your `snmpd.conf` file, this tag is required. See the `snmp_config(5)` manual page for the meaning

of this context switch.

[snmp] localCert <specifier>

This token defines the default X.509 public key to use as the server's identity. It should either be a fingerprint or a filename. To create a public key for use, please run the "net-snmp-cert" utility which will help you create the required certificate.

The default value for this is the certificate in the "snmpd" named certificate file.

[snmp] tlsAlgorithms <algorithms>

This string will select the algorithms to use when negotiating security during (D)TLS session establishment. See the openssl manual page ciphers(1) for details on the format. Examples strings include:

DEFAULT  
ALL  
HIGH  
HIGH:!AES128-SHA

The default value is whatever openssl itself was configured with.

[snmp] x509CRLFile

If you are using a Certificate Authority (CA) that publishes a Certificate Revocation List (CRL) then this token can be used to specify the location in the filesystem of a copy of the CRL file. Note that Net-SNMP will not pull a CRL over http and this must be a file, not a URL. Additionally, OpenSSL does not reload a CRL file when it has changed so modifications or updates to the file will only be noticed upon a restart of the snmpd agent.

certSecName PRIORITY FINGERPRINT OPTIONS

OPTIONS can be one of <--sn SECNAME | --rfc822 | --dns | --ip | --cn | --any>.

The certSecName token will specify how to map a certificate field from the client's X.509 certificate to a SNMPv3 username. Use the --sn SECNAME flag to directly specify a securityName for a given certificate. The other flags extract a particular component of the certificate for use as a snmpv3 securityName. These fields are one of: A SubjectAltName containing an rfc822 value (eg hardaker@net-snmp.org), A SubjectAltName containing a dns name value (eg foo.net-snmp.org), an IP address (eg 192.0.2.1) or a common name "Wes Hardaker". The --any flag specifies that any of the subjecAltName fields may be used. Make sure once a securityName has been selected that it is given authorization via the VACM controls discussed later in this manual page.

See the [http://www.net-snmp.org/wiki/index.php/Using\\_DTLS](http://www.net-snmp.org/wiki/index.php/Using_DTLS) web page for more detailed instructions for setting up (D)TLS.

trustCert <specifier>

For X509 to properly verify a certificate, it should be verifiable up until a trust anchor for it. This trust anchor is typically a CA certificate but it could also be a self-signed certificate. The "trustCert" token should be used to load specific trust anchors into the verification engine.

SNMP over (D)TLS requires the use of the Transport Security Model (TSM), so read the section on the usage of the Transport Security Model as well. Make sure when you configure the VACM to accept connections from (D)TLS that you use the "tsm" security model. E.G.:

```
rwuser -s tsm hardaker@net-snmp.org
```

### SNMPv3 over SSH Support

To use SSH, you'll need to configure sshd to invoke the sstosnmp program as well as configure the access control settings to allow access through the tsm security model using the user name provided to snmpd by the ssh transport.

### SNMPv3 with the Transport Security Model (TSM)

The Transport Security Model [RFC5591] defines a SNMPv3 security system for use with "tunneled" security protocols like TLS, DTLS and SSH. It is a very simple security model that simply lets properly protected packets to pass through into the snmp application. The transport is required to pass a securityName to use to the TSM and the TSM may optionally prefix this with a transport string (see below).

tsmUseTransportPrefix (1|yes|true|0|no|false)

If set to true, the TSM module will take every securityName passed to it from the transports underneath and prefix it with a string that specifically identifies the transport it came from. This is useful to avoid securityName clashes with transports that generate identical security names. For example, if the ssh security transport delivered the security name of "hardaker" for a SSH connection and the TLS security transport also delivered the security name of "hardaker" for a TLS connection then it would be impossible to separate out these two users to provide separate access control rights. With the tsmUseTransportPrefix set to true, however, the securityNames would be prefixed appropriately with one of: "tls:", "dtls:" or "ssh:".

### SNMPv3 with the User-based Security Model (USM)

SNMPv3 was originally defined using the User-Based Security Model (USM), which contains a private list of users and keys specific to the SNMPv3 protocol. The operational community, however, declared it a pain to manipulate yet another database and would prefer to use existing infrastructure. To that end the IETF created the ISMS working group to battle that problem, and the ISMS working group decided to tunnel SNMP over SSH and DTLS to make use existing user and authentication infrastructures.

#### SNMPv3 USM Users

To use the USM based SNMPv3-specific users, you'll need to create them. It is recommended you **use the net-snmp-config command** to do this, but you can also do it by directly specifying createUser directives yourself instead:

```
createUser [-e ENGINEID] username (MD5|SHA) authpassphrase [DES|AES] [privpassphrase]
```

MD5 and SHA are the authentication types to use. DES and AES are the privacy protocols to use. If the privacy passphrase is not specified, it is assumed to be the same as the authentication passphrase. Note that the users created will be useless unless they are also added to the VACM access control tables described above.

SHA authentication and DES/AES privacy require OpenSSL to be installed and the agent to be built with OpenSSL support. MD5 authentication may be used without OpenSSL.

Warning: the minimum pass phrase length is 8 characters.

SNMPv3 users can be created at runtime using the *snmpusm(1)* command.

Instead of figuring out how to use this directive and where to put it (see below), just run "net-snmp-config --create-snmpv3-user" instead, which will add one of these lines to the right place.

This directive should be placed into the /var/net-snmp/snmpd.conf file instead of the other normal locations. The reason is that the information is read from the file and then the line is removed (eliminating the storage of the master password for that user) and replaced with the key that is derived from it. This key is a localized key, so that if it is stolen it can not be used to access other agents. If the password is stolen, however, it can be.

If you need to localize the user to a particular EngineID (this is useful mostly in the similar snmpttrapd.conf file), you can use the -e argument to specify an EngineID as a hex value (EG, "0x01020304").

If you want to generate either your master or localized keys directly, replace the given password with a hexstring (preceded by a "0x") and precede the hex string by a -m or -l token (respectively). EGs:

[these keys are *\*not\** secure but are easy to visually parse for counting purposes. Please generate random keys instead of using

these examples]

```
createUser myuser SHA -l 0x0001020304050607080900010203040506070809 AES -l 0x00010203040506070809
createUser myuser SHA -m 0x0001020304050607080900010203040506070809 AES -m 0x00010203040506070809
```

Due to the way localization happens, localized privacy keys are expected to be the length needed by the algorithm (128 bits for all supported algorithms). Master encryption keys, though, need to be the length required by the authentication algorithm not the length required by the encrypting algorithm (MD5: 16 bytes, SHA: 20 bytes).

## ACCESS CONTROL

**snmpd** supports the View-Based Access Control Model (VACM) as defined in RFC 2575, to control who can retrieve or update information. To this end, it recognizes various directives relating to access control.

### Traditional Access Control

Most simple access control requirements can be specified using the directives *rouser/rwuser* (for SNMPv3) or *rocommunity/rwcommunity* (for SNMPv1 or SNMPv2c).

**rouser** [-s SECMODEL] USER [noauth|auth|priv [OID | -V VIEW [CONTEXT]]]

**rwuser** [-s SECMODEL] USER [noauth|auth|priv [OID | -V VIEW [CONTEXT]]]

specify an SNMPv3 user that will be allowed read-only (GET and GETNEXT) or read-write (GET, GETNEXT and SET) access respectively. By default, this will provide access to the full OID tree for authenticated (including encrypted) SNMPv3 requests, using the default context. An alternative minimum security level can be specified using *noauth* (to allow unauthenticated requests), or *priv* (to enforce use of encryption). The OID field restricts access for that user to the subtree rooted at the given OID, or the named view. An optional context can also be specified, or "context\*" to denote a context prefix. If no context field is specified (or the token "\*" is used), the directive will match all possible contexts.

If SECMODEL is specified then it will be the security model required for that user (note that identical user names may come in over different security models and will be appropriately separated via the access control settings). The default security model is "usm" and the other common security models are likely "tsm" when using (D)TLS or SSH support and "ksm" if the Kerberos support has been compiled in.

**rocommunity** COMMUNITY [SOURCE [OID | -V VIEW [CONTEXT]]]

**rwcommunity** COMMUNITY [SOURCE [OID | -V VIEW [CONTEXT]]]

specify an SNMPv1 or SNMPv2c community that will be allowed read-only (GET and GETNEXT) or read-write (GET, GETNEXT and SET) access respectively. By default, this will provide access to the full OID tree for such requests, regardless of where they were sent from. The SOURCE token can be used to restrict access to requests from the specified system(s) - see *com2sec* for the full details. The OID field restricts access for that community to the subtree rooted at the given OID, or named view. Contexts are typically less relevant to community-based SNMP versions, but the same behaviour applies here.

**rocommunity6** COMMUNITY [SOURCE [OID | -V VIEW [CONTEXT]]]

**rwcommunity6** COMMUNITY [SOURCE [OID | -V VIEW [CONTEXT]]]

are directives relating to requests received using IPv6 (if the agent supports such transport domains). The interpretation of the SOURCE, OID, VIEW and CONTEXT tokens are exactly the same as for the IPv4 versions.

In each case, only one directive should be specified for a given SNMPv3 user, or community string. It is **not** appropriate to specify both *rouser* and *rwuser* directives referring to the same SNMPv3 user (or equivalent community settings). The *rwuser* directive provides all the access of *rouser* (as well as allowing SET support). The same holds true for the community-based directives.

More complex access requirements (such as access to two or more distinct OID subtrees, or different views for GET and SET requests) should use one of the other access control mechanisms. Note that if several distinct communities or SNMPv3 users need to be granted the same level of access, it would also be more

efficient to use the main VACM configuration directives.

### VACM Configuration

The full flexibility of the VACM is available using four configuration directives – *com2sec*, *group*, *view* and *access*. These provide direct configuration of the underlying VACM tables.

**com2sec** [-Cn CONTEXT] SECNAME SOURCE COMMUNITY

**com2sec6** [-Cn CONTEXT] SECNAME SOURCE COMMUNITY

map an SNMPv1 or SNMPv2c community string to a security name - either from a particular range of source addresses, or globally ("*default*"). A restricted source can either be a specific host-name (or address), or a subnet - represented as IP/MASK (e.g. 10.10.10.0/255.255.255.0), or IP/BITS (e.g. 10.10.10.0/24), or the IPv6 equivalents.

The same community string can be specified in several separate directives (presumably with different source tokens), and the first source/community combination that matches the incoming request will be selected. Various source/community combinations can also map to the same security name.

If a CONTEXT is specified (using -Cn), the community string will be mapped to a security name in the named SNMPv3 context. Otherwise the default context ("") will be used.

**com2secunix** [-Cn CONTEXT] SECNAME SOCKPATH COMMUNITY

is the Unix domain sockets version of *com2sec*.

**group** GROUP {v1|v2c|usm|tsm|ksm} SECNAME

maps a security name (in the specified security model) into a named group. Several *group* directives can specify the same group name, allowing a single access setting to apply to several users and/or community strings.

Note that groups must be set up for the two community-based models separately - a single *com2sec* (or equivalent) directive will typically be accompanied by **two** *group* directives.

**view** VNAME TYPE OID [MASK]

defines a named "view" - a subset of the overall OID tree. This is most commonly a single subtree, but several *view* directives can be given with the same view name (VNAME), to build up a more complex collection of OIDs. TYPE is either *included* or *excluded*, which can again define a more complex view (e.g by excluding certain sensitive objects from an otherwise accessible subtree).

MASK is a list of hex octets (optionally separated by '.' or ':') with the set bits indicating which subidentifiers in the view OID to match against. If not specified, this defaults to matching the OID exactly (all bits set), thus defining a simple OID subtree. So:

```
view iso1 included .iso 0xf0
view iso2 included .iso
view iso3 included .iso.org.dod.mgmt 0xf0
```

would all define the same view, covering the whole of the 'iso(1)' subtree (with the third example ignoring the subidentifiers not covered by the mask).

More usefully, the mask can be used to define a view covering a particular row (or rows) in a table, by matching against the appropriate table index value, but skipping the column subidentifier:

```
view ifRow4 included .1.3.6.1.2.1.2.2.1.0.4 0xff:a0
```

Note that a mask longer than 8 bits must use ':' to separate the individual octets.

**access** GROUP CONTEXT {any|v1|v2c|usm|tsm|ksm} LEVEL PREFIX READ WRITE NOTIFY

maps from a group of users/communities (with a particular security model and minimum security level, and in a specific context) to one of three views, depending on the request being processed.

LEVEL is one of *noauth*, *auth*, or *priv*. PREFIX specifies how CONTEXT should be matched against the context of the incoming request, either *exact* or *prefix*. READ, WRITE and NOTIFY specifies the view to be used for GET\*, SET and TRAP/INFORM requests (although the NOTIFY view is not currently used). For v1 or v2c access, LEVEL will need to be *noauth*.

### Typed-View Configuration

The final group of directives extend the VACM approach into a more flexible mechanism, which can be applied to other access control requirements. Rather than the fixed three views of the standard VACM mechanism, this can be used to configure various different view types. As far as the main SNMP agent is concerned, the two main view types are *read* and *write*, corresponding to the READ and WRITE views of the main *access* directive. See the 'snmptrapd.conf(5)' man page for discussion of other view types.

**authcommunity** TYPES COMMUNITY [SOURCE [OID | -V VIEW [CONTEXT]]]

is an alternative to the *rocommunity/rwcommunity* directives. TYPES will usually be *read* or *read,write* respectively. The view specification can either be an OID subtree (as before), or a named view (defined using the *view* directive) for greater flexibility. If this is omitted, then access will be allowed to the full OID tree. If CONTEXT is specified, access is configured within this SNMPv3 context. Otherwise the default context ("" ) is used.

**authuser** TYPES [-s MODEL] USER [LEVEL [OID | -V VIEW [CONTEXT]]]

is an alternative to the *rouser/rwuser* directives. The fields TYPES, OID, VIEW and CONTEXT have the same meaning as for *authcommunity*.

**authgroup** TYPES [-s MODEL] GROUP [LEVEL [OID | -V VIEW [CONTEXT]]]

is a companion to the *authuser* directive, specifying access for a particular group (defined using the *group* directive as usual). Both *authuser* and *authgroup* default to authenticated requests - LEVEL can also be specified as *noauth* or *priv* to allow unauthenticated requests, or require encryption respectively. Both *authuser* and *authgroup* directives also default to configuring access for SNMPv3/USM requests - use the '-s' flag to specify an alternative security model (using the same values as for *access* above).

**authaccess** TYPES [-s MODEL] GROUP VIEW [LEVEL [CONTEXT]]

also configures the access for a particular group, specifying the name and type of view to apply. The MODEL and LEVEL fields are interpreted in the same way as for *authgroup*. If CONTEXT is specified, access is configured within this SNMPv3 context (or contexts with this prefix if the CONTEXT field ends with '\*'). Otherwise the default context ("" ) is used.

**setaccess** GROUP CONTEXT MODEL LEVEL PREFIX VIEW TYPES

is a direct equivalent to the original *access* directive, typically listing the view types as *read* or *read,write* as appropriate. (or see 'snmptrapd.conf(5)' for other possibilities). All other fields have the same interpretation as with *access*.

### SYSTEM INFORMATION

Most of the information reported by the Net-SNMP agent is retrieved from the underlying system, or dynamically configured via SNMP SET requests (and retained from one run of the agent to the next). However, certain MIB objects can be configured or controlled via the *snmpd.conf(5)* file.

#### System Group

Most of the scalar objects in the 'system' group can be configured in this way:

**sysLocation** STRING

**sysContact** STRING

**sysName** STRING

set the system location, system contact or system name (*sysLocation.0*, *sysContact.0* and *sysName.0*) for the agent respectively. Ordinarily these objects are writable via suitably authorized SNMP SET requests. However, specifying one of these directives makes the corresponding object read-only, and attempts to SET it will result in a *notWritable* error response.

**sysServices** NUMBER

sets the value of the *sysServices.0* object. For a host system, a good value is 72 (application + end-to-end layers). If this directive is not specified, then no value will be reported for the *sysServices.0* object.

sysDescr STRING

sysObjectID OID

sets the system description or object ID for the agent. Although these MIB objects are not SNMP-writable, these directives can be used by a network administrator to configure suitable values for them.

### Interfaces Group

interface NAME TYPE SPEED

can be used to provide appropriate type and speed settings for interfaces where the agent fails to determine this information correctly. TYPE is a type value as given in the IANAifType-MIB, and can be specified numerically or by name (assuming this MIB is loaded).

interface\_fadeout TIMEOUT

specifies, for how long the agent keeps entries in `ifTable` after appropriate interfaces have been removed from system (typically various ppp, tap or tun interfaces). Timeout value is in seconds. Default value is 300 (=5 minutes).

interface\_replace\_old yes

can be used to remove already existing entries in `ifTable` when an interface with the same name appears on the system. E.g. when ppp0 interface is removed, it is still listed in the table for *interface\_fadeout* seconds. This option ensures, that the old ppp0 interface is removed even before the *interface\_fadeout* timeout when new ppp0 (with different `ifIndex`) shows up.

### Host Resources Group

This requires that the agent was built with support for the *host* module (which is now included as part of the default build configuration on the major supported platforms).

ignoreDisk STRING

controls which disk devices are scanned as part of populating the `hrDiskStorageTable` (and `hrDeviceTable`). The HostRes implementation code includes a list of disk device patterns appropriate for the current operating system, some of which may cause the agent to block when trying to open the corresponding disk devices. This might lead to a timeout when walking these tables, possibly resulting in inconsistent behaviour. This directive can be used to specify particular devices (either individually or wildcarded) that should not be checked.

Note: Please consult the source (*host/hr\_disk.c*) and check for the *Add\_HR\_Disk\_entry* calls relevant for a particular O/S to determine the list of devices that will be scanned.

The pattern can include one or more wildcard expressions. See *snmpd.examples(5)* for illustration of the wildcard syntax.

skipNFSInHostResources true

controls whether NFS and NFS-like file systems should be omitted from the `hrStorageTable` (true or 1) or not (false or 0, which is the default). If the Net-SNMP agent gets hung on NFS-mounted filesystems, you can try setting this to '1'.

storageUseNFS [1|2]

controls how NFS and NFS-like file systems should be reported in the `hrStorageTable`. as 'Network Disks' (1) or 'Fixed Disks' (2) Historically, the Net-SNMP agent has reported such file systems as 'Fixed Disks', and this is still the default behaviour. Setting this directive to '1' reports such file systems as 'Network Disks', as required by the Host Resources MIB.

realStorageUnits

controls how the agent reports `hrStorageAllocationUnits`, `hrStorageSize` and `hrStorageUsed` in `hrStorageTable`. For big storage drives with small allocation units the agent re-calculates these values so they all fit Integer32 and `hrStorageAllocationUnits` x `hrStorageSize` gives real size of the storage.

Example:

Linux xfs 16TB filesystem with 4096 bytes large blocks will be reported as `hrStorageAllocationUnits` = 8192 and `hrStorageSize` = 2147483647, so 8192 x 2147483647 gives real



size of the filesystem (=16 TB).

Setting this directive to '1' turns off this calculation and the agent reports real `hrStorageAllocationUnits`, but it might report wrong `hrStorageSize` for big drives because the value won't fit into `Integer32`. In this case, `hrStorageAllocationUnits` x `hrStorageSize` won't give real size of the storage.

### Process Monitoring

The `hrSWRun` group of the Host Resources MIB provides information about individual processes running on the local system. The `prTable` of the UCD-SNMP-MIB complements this by reporting on selected services (which may involve multiple processes). This requires that the agent was built with support for the `ucd-smmp/proc` module (which is included as part of the default build configuration).

`proc NAME [MAX [MIN]]`

monitors the number of processes called NAME (as reported by `/bin/ps -e`) running on the local system.

If the number of NAMED processes is less than MIN or greater than MAX, then the corresponding `prErrorFlag` instance will be set to 1, and a suitable description message reported via the `prErrorMessage` instance.

Note: This situation will **not** automatically trigger a trap to report the problem - see the DisMan Event MIB section later.

If neither MAX nor MIN are specified, they will default to **infinity** and 1 respectively ("at least one"). If only MAX is specified, MIN will default to 0 ("no more than MAX"). If MAX is 0 (and MIN is not), this indicates infinity ("at least MIN"). If both MAX and MIN are 0, this indicates a process that should **not** be running.

`procfix NAME PROG ARGS`

registers a command that can be run to fix errors with the given process NAME. This will be invoked when the corresponding `prErrFix` instance is set to 1.

Note: This command will **not** be invoked automatically.

The *procfix* directive must be specified **after** the matching *proc* directive, and cannot be used on its own.

If no *proc* directives are defined, then walking the `prTable` will fail (*noSuchObject*).

### Disk Usage Monitoring

This requires that the agent was built with support for the `ucd-smmp/disk` module (which is included as part of the default build configuration).

`disk PATH [ MINSIZE | MINPERCENT% ]`

monitors the disk mounted at PATH for available disk space.

The minimum threshold can either be specified in kB (MINSIZE) or as a percentage of the total disk (MINPERCENT% with a '%' character), defaulting to 100kB if neither are specified. If the free disk space falls below this threshold, then the corresponding `dskErrorFlag` instance will be set to 1, and a suitable description message reported via the `dskErrorMsg` instance.

Note: This situation will **not** automatically trigger a trap to report the problem - see the DisMan Event MIB section later.

`includeAllDisks MINPERCENT%`

configures monitoring of all disks found on the system, using the specified (percentage) threshold. The threshold for individual disks can be adjusted using suitable *disk* directives (which can come either before or after the *includeAllDisks* directive).

Note: Whether *disk* directives appears before or after *includeAllDisks* may affect the indexing of the `dskTable`.

Only one *includeAllDisks* directive should be specified - any subsequent copies will be ignored.

The list of mounted disks will be determined when the agent starts using the `setmntent(3)` and `getmntent(3)`, or `fopen(3)` and `getmntent(3)`, or `setfsent(3)` and `getfsent(3)` system calls. If none of the above system calls are available then the root partition `"/"` (which is assumed to exist on any UNIX based system) will be monitored. Disks mounted after the agent has started will not be monitored.

If neither any *disk* directives or *includeAllDisks* are defined, then walking the `dskTable` will fail (*noSuchObject*).

### Disk I/O Monitoring

This requires that the agent was built with support for the *ucd-snmp/diskio* module (which is not included as part of the default build configuration).

By default, all block devices known to the operating system are included in the `diskIOTable`. On platforms other than Linux, this module has no configuration directives.

On Linux systems, it is possible to exclude several classes of block devices from the `diskIOTable` in order to avoid cluttering the table with useless zero statistics for pseudo-devices that often are not in use but are configured by default to exist in most recent Linux distributions.

`diskio_exclude_fd` yes

Excludes all Linux floppy disk block devices, whose names start with "fd", e.g. "fd0"

`diskio_exclude_loop` yes

Excludes all Linux loopback block devices, whose names start with "loop", e.g. "loop0"

`diskio_exclude_ram` yes

Excludes all Linux ramdisk block devices, whose names start with "ram", e.g. "ram0"

### System Load Monitoring

This requires that the agent was built with support for either the *ucd-snmp/loadave* module or the *ucd-snmp/memory* module respectively (both of which are included as part of the default build configuration).

`load MAX1 [MAX5 [MAX15]]`

monitors the load average of the local system, specifying thresholds for the 1-minute, 5-minute and 15-minute averages. If any of these loads exceed the associated maximum value, then the corresponding `laErrorFlag` instance will be set to 1, and a suitable description message reported via the `laErrorMessage` instance.

Note: This situation will **not** automatically trigger a trap to report the problem - see the `DisMan` Event MIB section later.

If the `MAX15` threshold is omitted, it will default to the `MAX5` value. If both `MAX5` and `MAX15` are omitted, they will default to the `MAX1` value. If this directive is not specified, all three thresholds will default to a value of `DEFMAXLOADAVE`.

If a threshold value of 0 is given, the agent will not report errors via the relevant `laErrorFlag` or `laErrorMessage` instances, regardless of the current load.

Unlike the *proc* and *disk* directives, walking the `laTable` will succeed (assuming the *ucd-snmp/loadave* module was configured into the agent), even if the *load* directive is not present.

`swap MIN`

monitors the amount of swap space available on the local system. If this falls below the specified threshold (`MIN` kB), then the `memErrorSwap` object will be set to 1, and a suitable description message reported via `memSwapErrorMsg`.

Note: This situation will **not** automatically trigger a trap to report the problem - see the `DisMan` Event MIB section later.

If this directive is not specified, the default threshold is 16 MB.

### Log File Monitoring

This requires that the agent was built with support for either the *ucd-smnp/file* or *ucd-smnp/logmatch* modules respectively (both of which are included as part of the default build configuration).

**file** FILE [MAXSIZE]

monitors the size of the specified file (in kB). If MAXSIZE is specified, and the size of the file exceeds this threshold, then the corresponding `fileErrorFlag` instance will be set to 1, and a suitable description message reported via the `fileErrorMsg` instance.

Note: This situation will **not** automatically trigger a trap to report the problem - see the DisMan Event MIB section later.

Note: A maximum of 20 files can be monitored.

Note: If no *file* directives are defined, then walking the `fileTable` will fail (*noSuchObject*).

**logmatch** NAME FILE CYCLETIME REGEX

monitors the specified file for occurrences of the specified pattern REGEX. The file position is stored internally so the entire file is only read initially, every subsequent pass will only read the new lines added to the file since the last read.

**NAME** name of the logmatch instance (will appear as `logMatchName` under `logMatch/logMatchTable/logMatchEntry/logMatchName` in the *ucd-smnp* MIB tree)

**FILE** absolute path to the logfile to be monitored. Note that this path can contain date/time directives (like in the UNIX 'date' command). See the manual page for 'strftime' for the various directives accepted.

**CYCLETIME**

time interval for each logfile read and internal variable update in seconds. Note: an `SNMPGET*` operation will also trigger an immediate logfile read and variable update.

**REGEX**

the regular expression to be used. Note: DO NOT enclose the regular expression in quotes even if there are spaces in the expression as the quotes will also become part of the pattern to be matched!

Example:

```
logmatch    apache-GETs    /usr/local/apache/logs/access.log-%Y-%m-%d    60
GET.*HTTP.*
```

This logmatch instance is named 'apache-GETs', uses 'GET.\*HTTP.\*' as its regular expression and it will monitor the file named (assuming today is May 6th 2009): '/usr/local/apache/logs/access.log-2009-05-06', tomorrow it will look for 'access.log-2009-05-07'. The logfile is read every 60 seconds.

Note: A maximum of 250 logmatch directives can be specified.

Note: If no *logmatch* directives are defined, then walking the `logMatchTable` will fail (*noSuchObject*).

### ACTIVE MONITORING

The usual behaviour of an SNMP agent is to wait for incoming SNMP requests and respond to them - if no requests are received, an agent will typically not initiate any actions. This section describes various directives that can configure *snmpd* to take a more active role.

#### Notification Handling

**trapcommunity** STRING

defines the default community string to be used when sending traps. Note that this directive must be used prior to any community-based trap destination directives that need to use it.

**trapsink** HOST [COMMUNITY [PORT]]

trap2sink HOST [COMMUNITY [PORT]]

informsink HOST [COMMUNITY [PORT]]

define the address of a notification receiver that should be sent SNMPv1 TRAPs, SNMPv2c TRAP2s, or SNMPv2 INFORM notifications respectively. See the section **LISTENING ADDRESSES** in the *snmpd(8)* manual page for more information about the format of listening addresses. If COMMUNITY is not specified, the most recent *trapcommunity* string will be used.

If the transport address does not include an explicit port specification, then PORT will be used. If this is not specified, the well known SNMP trap port (162) will be used.

Note: This mechanism is being deprecated, and the listening port should be specified via the transport specification HOST instead.

If several sink directives are specified, multiple copies of each notification (in the appropriate formats) will be generated.

Note: It is **not** normally appropriate to list two (or all three) sink directives with the same destination.

trapssess [SNMPCMD\_ARGS] HOST

provides a more generic mechanism for defining notification destinations. *SNMPCMD\_ARGS* should be the command-line options required for an equivalent *snmptrap* (or *snmpinform*) command to send the desired notification. The option *-Ci* can be used (with *-v2c* or *-v3*) to generate an INFORM notification rather than an unacknowledged TRAP.

This is the appropriate directive for defining SNMPv3 trap receivers. See <http://www.net-snmp.org/tutorial/tutorial-5/commands/snmptrap-v3.html> for more information about SNMPv3 notification behaviour.

authtrapenable {1|2}

determines whether to generate authentication failure traps (*enabled(1)*) or not (*disabled(2)* - the default). Ordinarily the corresponding MIB object (*snmpEnableAuthenTraps.0*) is read-write, but specifying this directive makes this object read-only, and attempts to set the value via SET requests will result in a *notWritable* error response.

v1trapaddress HOST

defines the agent address, which is inserted into SNMPv1 TRAPs. Arbitrary local IPv4 address is chosen if this option is omitted. This option is useful mainly when the agent is visible from outside world by specific address only (e.g. because of network address translation or firewall).

## DisMan Event MIB

The previous directives can be used to configure where traps should be sent, but are not concerned with *when* to send such traps (or what traps should be generated). This is the domain of the Event MIB - developed by the Distributed Management (DisMan) working group of the IETF.

This requires that the agent was built with support for the *disman/event* module (which is now included as part of the default build configuration for the most recent distribution).

Note: The behaviour of the latest implementation differs in some minor respects from the previous code - nothing too significant, but existing scripts may possibly need some minor adjustments.

iquerySecName NAME

agentSecName NAME

specifies the default SNMPv3 username, to be used when making internal queries to retrieve any necessary information (either for evaluating the monitored expression, or building a notification payload). These internal queries always use SNMPv3, even if normal querying of the agent is done using SNMPv1 or SNMPv2c.

Note that this user must also be explicitly created (*createUser*) and given appropriate access rights (e.g. *rouser*). This directive is purely concerned with defining *which* user should be used - not

with actually setting this user up.

#### monitor [OPTIONS] NAME EXPRESSION

defines a MIB object to monitor. If the EXPRESSION condition holds (see below), then this will trigger the corresponding event, and either send a notification or apply a SET assignment (or both). Note that the event will only be triggered once, when the expression first matches. This monitor entry will not fire again until the monitored condition first becomes false, and then matches again. NAME is an administrative name for this expression, and is used for indexing the `mteTriggerTable` (and related tables). Note also that such monitors use an internal SNMPv3 request to retrieve the values being monitored (even if normal agent queries typically use SNMPv1 or SNMPv2c). See the *iquerySecName* token described above.

#### EXPRESSION

There are three types of monitor expression supported by the Event MIB - existence, boolean and threshold tests.

##### OID | ! OID | != OID

defines an *existence(0)* monitor test. A bare OID specifies a *present(0)* test, which will fire when (an instance of) the monitored OID is created. An expression of the form *! OID* specifies an *absent(1)* test, which will fire when the monitored OID is deleted. An expression of the form *!= OID* specifies a *changed(2)* test, which will fire whenever the monitored value(s) change. Note that there **must** be whitespace before the OID token.

##### OID OP VALUE

defines a *boolean(1)* monitor test. OP should be one of the defined comparison operators (*!=*, *==*, *<*, *<=*, *>*, *>=*) and VALUE should be an integer value to compare against. Note that there **must** be whitespace around the OP token. A comparison such as *OID !=0* will not be handled correctly.

##### OID MIN MAX [DMIN DMAX]

defines a *threshold(2)* monitor test. MIN and MAX are integer values, specifying lower and upper thresholds. If the value of the monitored OID falls below the lower threshold (MIN) or rises above the upper threshold (MAX), then the monitor entry will trigger the corresponding event.

Note that the rising threshold event will only be re-armed when the monitored value falls below the **lower** threshold (MIN). Similarly, the falling threshold event will be re-armed by the upper threshold (MAX).

The optional parameters DMIN and DMAX configure a pair of similar threshold tests, but working with the delta differences between successive sample values.

#### OPTIONS

There are various options to control the behaviour of the monitored expression. These include:

–D indicates that the expression should be evaluated using delta differences between sample values (rather than the values themselves).

–d OID

–di OID

specifies a discontinuity marker for validating delta differences. A *–di* object instance will be used exactly as given. A *–d* object will have the instance subidentifiers from the corresponding (wildcarded) expression object appended. If the *–I* flag is specified, then there is no difference between these two options.

This option also implies *–D*.

–e EVENT

specifies the event to be invoked when this monitor entry is triggered. If this option is not given, the monitor entry will generate one of the standard notifications defined in the DISMAN–EVENT–MIB.

–I indicates that the monitored expression should be applied to the specified OID as a single instance. By default, the OID will be treated as a wildcarded object, and the monitor expanded to cover all matching instances.

–i OID

–o OID define additional varbinds to be added to the notification payload when this monitor trigger fires. For a wildcarded expression, the suffix of the matched instance will be added to any OIDs specified using –o, while OIDs specified using –i will be treated as exact instances. If the –I flag is specified, then there is no difference between these two options.

See *strictDisman* for details of the ordering of notification payloads.

–r FREQUENCY

monitors the given expression every FREQUENCY, where FREQUENCY is in seconds or optionally suffixed by one of s (for seconds), m (for minutes), h (for hours), d (for days), or w (for weeks). By default, the expression will be evaluated every 600s (10 minutes).

–S indicates that the monitor expression should *not* be evaluated when the agent first starts up. The first evaluation will be done once the first repeat interval has expired.

–s indicates that the monitor expression *should* be evaluated when the agent first starts up. This is the default behaviour.

Note: Notifications triggered by this initial evaluation will be sent *before* the cold-start trap.

–u SECNAME

specifies a security name to use for scanning the local host, instead of the default *iquery-SecName*. Once again, this user must be explicitly created and given suitable access rights.

notificationEvent ENAME NOTIFICATION [–m] [–i OID | –o OID ]\*

defines a notification event named ENAME. This can be triggered from a given *monitor* entry by specifying the option –e ENAME (see above). NOTIFICATION should be the OID of the NOTIFICATION-TYPE definition for the notification to be generated.

If the –m option is given, the notification payload will include the standard varbinds as specified in the OBJECTS clause of the notification MIB definition. This option must come **after** the NOTIFICATION OID (and the relevant MIB file must be available and loaded by the agent). Otherwise, these varbinds must be listed explicitly (either here or in the corresponding *monitor* directive).

The –i OID and –o OID options specify additional varbinds to be appended to the notification payload, after the standard list. If the monitor entry that triggered this event involved a wildcarded expression, the suffix of the matched instance will be added to any OIDs specified using –o, while OIDs specified using –i will be treated as exact instances. If the –I flag was specified to the *monitor* directive, then there is no difference between these two options.

setEvent ENAME [–I] OID = VALUE

defines a set event named ENAME, assigning the (integer) VALUE to the specified OID. This can be triggered from a given *monitor* entry by specifying the option –e ENAME (see above).

If the monitor entry that triggered this event involved a wildcarded expression, the suffix of the matched instance will normally be added to the OID. If the –I flag was specified to either of the *monitor* or *setEvent* directives, the specified OID will be regarded as an exact single instance.

strictDisman yes

The definition of SNMP notifications states that the varbinds defined in the OBJECT clause should come first (in the order specified), followed by any "extra" varbinds that the notification generator feels might be useful. The most natural approach would be to associate these mandatory varbinds with the *notificationEvent* entry, and append any varbinds associated with the monitor entry that

triggered the notification to the end of this list. This is the default behaviour of the Net-SNMP Event MIB implementation.

Unfortunately, the DisMan Event MIB specifications actually state that the trigger-related varbinds should come **first**, followed by the event-related ones. This directive can be used to restore this strictly-correct (but inappropriate) behaviour.

Note: Strict DisMan ordering may result in generating invalid notifications payload lists if the *notificationEvent -n* flag is used together with *monitor -o* (or *-i*) varbind options.

If no *monitor* entries specify payload varbinds, then the setting of this directive is irrelevant.

**linkUpDownNotifications yes**

will configure the Event MIB tables to monitor the `ifTable` for network interfaces being taken up or down, and triggering a *linkUp* or *linkDown* notification as appropriate.

This is exactly equivalent to the configuration:

```
notificationEvent linkUpTrap linkUp ifIndex ifAdminStatus ifOperStatus
notificationEvent linkDownTrap linkDown ifIndex ifAdminStatus ifOperStatus

monitor -r 60 -e linkUpTrap "Generate linkUp" ifOperStatus != 2
monitor -r 60 -e linkDownTrap "Generate linkDown" ifOperStatus == 2
```

**defaultMonitors yes**

will configure the Event MIB tables to monitor the various UCD-SNMP-MIB tables for problems (as indicated by the appropriate `xxErrFlag` column objects).

This is exactly equivalent to the configuration:

```
monitor -o prNames -o prErrMsg "process table" prErrorFlag != 0
monitor -o memErrorName -o memSwapErrorMsg "memory" memSwapError != 0
monitor -o extNames -o extOutput "extTable" extResult != 0
monitor -o dskPath -o dskErrorMsg "dskTable" dskErrorFlag != 0
monitor -o laNames -o laErrMsg "laTable" laErrorFlag != 0
monitor -o fileName -o fileErrorMsg "fileTable" fileErrorFlag != 0
```

In both these latter cases, the `snmpd.conf` must also contain a *iquerySecName* directive, together with a corresponding *createUser* entry and suitable access control configuration.

### DisMan Schedule MIB

The DisMan working group also produced a mechanism for scheduling particular actions (a specified SET assignment) at given times. This requires that the agent was built with support for the *disman/schedule* module (which is included as part of the default build configuration for the most recent distribution).

There are three ways of specifying the scheduled action:

**repeat FREQUENCY OID = VALUE**

configures a SET assignment of the (integer) VALUE to the MIB instance OID, to be run every FREQUENCY seconds, where FREQUENCY is in seconds or optionally suffixed by one of s (for seconds), m (for minutes), h (for hours), d (for days), or w (for weeks).

**cron MINUTE HOUR DAY MONTH WEEKDAY OID = VALUE**

configures a SET assignment of the (integer) VALUE to the MIB instance OID, to be run at the times specified by the fields MINUTE to WEEKDAY. These follow the same pattern as the equivalent *crontab(5)* fields.

Note: These fields should be specified as a (comma-separated) list of numeric values. Named values for the MONTH and WEEKDAY fields are not supported, and neither are value ranges. A wildcard match can be specified as '\*'.

The DAY field can also accept negative values, to indicate days counting backwards from the end of the month.

at MINUTE HOUR DAY MONTH WEEKDAY OID = VALUE

configures a one-shot SET assignment, to be run at the first matching time as specified by the fields MINUTE to WEEKDAY. The interpretation of these fields is exactly the same as for the *cron* directive.

### Data Delivery via Notifications

Note: this functionality is only available if the *deliver/deliverByNotify* mib module was compiled in to the agent

In some situations it may be advantageous to deliver SNMP data over SNMP Notifications (TRAPs and INFORMs) rather than the typical process of having the manager issue requests for the data (via GETs and GETNEXTs). Reasons for doing this are numerous, but frequently corner cases. The most common reason for wanting this behaviour might be to monitor devices that reside behind NATs or Firewalls that prevent incoming SNMP traffic.

It should be noted that although most management software is capable of logging notifications, very little (if any) management software will update their "knowledge database" based on the contents of SNMP notifications. IE, it won't (for example) update the interface traffic counter history that is used to produce graphs. Most larger network management packages have a separate database for storing data received via SNMP requests (GETs and GETNEXTs) vs those received from notifications. Researching the capabilities of your management station software is required before assuming this functionality will solve your data delivery requirements.

Notifications generated via this mechanism will be sent to the standard set of configured notification targets. See the "Notification Handling" section of this document for further information.

**deliverByNotify [-p] [-m] [-s MAXSIZE] FREQUENCY OID**

This directive tells the SNMP agent to self-walk the *OID*, collect all the data and send it out every *FREQUENCY* seconds, where *FREQUENCY* is in seconds or optionally suffixed by one of s (for seconds), m (for minutes), h (for hours), d (for days), or w (for weeks). By default scalars are included in the notification that specify the how often the notification will be sent (unless the *-p* option is specified) and which message number of how many messages a particular notification is (unless *-m* is specified). To break the notifications into manageable packet sizes, use the *-s* flag to specify the approximate maximum number of bytes that a notification message should be limited to. If more than *MAXSIZE* of bytes is needed then multiple notifications will be sent to deliver the data. Note that the calculations for ensuring the maximum size is met are approximations and thus it can be absolutely guaranteed they'll be under that size, so leave a padding buffer if it is critical that you avoid fragmentation. A value of *-1* indicates force everything into a single message no matter how big it is.

Example usage: the following will deliver the contents of the *ifTable* once an hour and the contents of the *system* group once every 2 hours:

```
deliverByNotify 3600 ifTable
deliverByNotify 7200 system
```

**deliverByNotifyMaxPacketSize SIZEINBYTES**

Sets the default notification size limit (see the *-s* flag above).

**deliverByNotifyOid OID**

**deliverByNotifyFrequencyOid OID**

**deliverByNotifyMessageNumberOid OID**

**deliverByNotifyMaxMessageNumberOid OID**

These set the data OID that the notification will be sent under, the scalar OID, the message number OID, and the maximum message number OID. These default to objects in the NET-SNMP-PERIODIC-NOTIFY-MIB.



## EXTENDING AGENT FUNCTIONALITY

One of the first distinguishing features of the original UCD suite was the ability to extend the functionality of the agent - not just by recompiling with code for new MIB modules, but also by configuring the running agent to report additional information. There are a number of techniques to support this, including:

- running external commands (*exec*, *extend*, *pass*)
- loading new code dynamically (embedded perl, *dlmod*)
- communicating with other agents (*proxy*, *SMUX*, *AgentX*)

### Arbitrary Extension Commands

The earliest extension mechanism was the ability to run arbitrary commands or shell scripts. Such commands do not need to be aware of SNMP operations, or conform to any particular behaviour - the MIB structures are designed to accommodate any form of command output. Use of this mechanism requires that the agent was built with support for the *ucd-smmp/extensible* and/or *agent/extend* modules (which are both included as part of the default build configuration).

*exec* [MIBOID] NAME PROG ARGS

*sh* [MIBOID] NAME PROG ARGS

invoke the named PROG with arguments of ARGS. By default the exit status and first line of output from the command will be reported via the `extTable`, discarding any additional output.

Note: Entries in this table appear in the order they are read from the configuration file. This means that adding new *exec* (or *sh*) directives and restarting the agent, may affect the indexing of other entries.

The PROG argument for *exec* directives must be a full path to a real binary, as it is executed via the `exec()` system call. To invoke a shell script, use the *sh* directive instead.

If MIBOID is specified, then the results will be rooted at this point in the OID tree, returning the exit statement as MIBOID.100.0 and the entire command output in a pseudo-table based at MIBNUM.101 - with one 'row' for each line of output.

Note: The layout of this "relocatable" form of *exec* (or *sh*) output does not strictly form a valid MIB structure. This mechanism is being deprecated - please see the *extend* directive (described below) instead.

The agent does not cache the exit status or output of the executed program.

*execfix* NAME PROG ARGS

registers a command that can be invoked on demand - typically to respond to or fix errors with the corresponding *exec* or *sh* entry. When the *extErrFix* instance for a given NAMED entry is set to the integer value of 1, this command will be called.

Note: This directive can only be used in combination with a corresponding *exec* or *sh* directive, which must be defined first. Attempting to define an unaccompanied *execfix* directive will fail.

*exec* and *sh* extensions can only be configured via the `snmpd.conf` file. They cannot be set up via SNMP SET requests.

*extend* [MIBOID] NAME PROG ARGS

works in a similar manner to the *exec* directive, but with a number of improvements. The MIB tables (*nsExtendConfigTable* etc) are indexed by the NAME token, so are unaffected by the order in which entries are read from the configuration files. There are *two* result tables - one (*nsExtendOutput1Table*) containing the exit status, the first line and full output (as a single string) for each *extend* entry, and the other (*nsExtendOutput2Table*) containing the complete output as a series of separate lines.

If MIBOID is specified, then the configuration and result tables will be rooted at this point in the OID tree, but are otherwise structured in exactly the same way. This means that several separate *extend* directives can specify the same MIBOID root, without conflicting.

The exit status and output is cached for each entry individually, and can be cleared (and the caching behaviour configured) using the `nsCacheTable`.

`extendfix NAME PROG ARGS`

registers a command that can be invoked on demand, by setting the appropriate `nsExtendRunType` instance to the value `run-command(3)`. Unlike the equivalent `execfix`, this directive does not need to be paired with a corresponding `extend` entry, and can appear on its own.

Both `extend` and `extendfix` directives can be configured dynamically, using SNMP SET requests to the `NET-SNMP-EXTEND-MIB`.

### MIB-Specific Extension Commands

The first group of extension directives invoke arbitrary commands, and rely on the MIB structure (and management applications) having the flexibility to accommodate and interpret the output. This is a convenient way to make information available quickly and simply, but is of no use when implementing specific MIB objects, where the extension must conform to the structure of the MIB (rather than vice versa). The remaining extension mechanisms are all concerned with such MIB-specific situations - starting with "pass-through" scripts. Use of this mechanism requires that the agent was built with support for the `ucd-smnp/pass` and `ucd-smnp/pass_persist` modules (which are both included as part of the default build configuration).

`pass [-p priority] MIBOID PROG`

will pass control of the subtree rooted at MIBOID to the specified PROG command. GET and GETNEXT requests for OIDs within this tree will trigger this command, called as:

`PROG -g OID`

`PROG -n OID`

respectively, where OID is the requested OID. The PROG command should return the response varbind as three separate lines printed to stdout - the first line should be the OID of the returned value, the second should be its TYPE (one of the text strings **integer**, **gauge**, **counter**, **timeticks**, **ipaddress**, **objectid**, or **string**), and the third should be the value itself.

If the command cannot return an appropriate varbind - e.g the specified OID did not correspond to a valid instance for a GET request, or there were no following instances for a GETNEXT - then it should exit without producing any output. This will result in an SNMP *noSuchName* error, or a *noSuchInstance* exception.

Note: The SMIV2 type **counter64** and SNMPv2 *noSuchObject* exception are not supported.

A SET request will result in the command being called as:

`PROG -s OID TYPE VALUE`

where TYPE is one of the tokens listed above, indicating the type of the value passed as the third parameter.

If the assignment is successful, the PROG command should exit without producing any output. Errors should be indicated by writing one of the strings **not-writable**, or **wrong-type** to stdout, and the agent will generate the appropriate error response.

Note: The other SNMPv2 errors are not supported.

In either case, the command should exit once it has finished processing. Each request (and each varbind within a single request) will trigger a separate invocation of the command.

The default registration priority is 127. This can be changed by supplying the optional `-p` flag, with lower priority registrations being used in preference to higher priority values.

`pass_persist [-p priority] MIBOID PROG`

will also pass control of the subtree rooted at MIBOID to the specified PROG command. However this command will continue to run after the initial request has been answered, so subsequent requests can be processed without the startup overheads.

Upon initialization, PROG will be passed the string "PING\n" on stdin, and should respond by printing "PONG\n" to stdout.

For GET and GETNEXT requests, PROG will be passed two lines on stdin, the command (*get* or *getnext*) and the requested OID. It should respond by printing three lines to stdout - the OID for the result varbind, the TYPE and the VALUE itself - exactly as for the *pass* directive above. If the command cannot return an appropriate varbind, it should print "NONE\n" to stdout (but continue running).

For SET requests, PROG will be passed three lines on stdin, the command (*set*) and the requested OID, followed by the type and value (both on the same line). If the assignment is successful, the command should print "DONE\n" to stdout. Errors should be indicated by writing one of the strings **not-writable**, **wrong-type**, **wrong-length**, **wrong-value** or **inconsistent-value** to stdout, and the agent will generate the appropriate error response. In either case, the command should continue running.

The registration priority can be changed using the optional *-p* flag, just as for the *pass* directive.

*pass* and *pass\_persist* extensions can only be configured via the *snmpd.conf* file. They cannot be set up via SNMP SET requests.

### Embedded Perl Support

Programs using the previous extension mechanisms can be written in any convenient programming language - including perl, which is a common choice for pass-through extensions in particular. However the Net-SNMP agent also includes support for embedded perl technology (similar to *mod\_perl* for the Apache web server). This allows the agent to interpret perl scripts directly, thus avoiding the overhead of spawning processes and initializing the perl system when a request is received.

Use of this mechanism requires that the agent was built with support for the embedded perl mechanism, which is not part of the default build environment. It must be explicitly included by specifying the *'--enable-embedded-perl'* option to the configure script when the package is first built.

If enabled, the following directives will be recognised:

*disablePerl* true

will turn off embedded perl support entirely (e.g. if there are problems with the perl installation).

*perlInitFile* FILE

loads the specified initialisation file (if present) immediately before the first *perl* directive is parsed. If not explicitly specified, the agent will look for the default initialisation file */code/git/mgt/target/install/share/snmp/snmp\_perl.pl*.

The default initialisation file creates an instance of a *NetSNMP::agent* object - a variable *\$agent* which can be used to register perl-based MIB handler routines.

*perl* EXPRESSION

evaluates the given expression. This would typically register a handler routine to be called when a section of the OID tree was requested:

```
perl use Data::Dumper;
perl sub myroutine { print "got called: ",Dumper(@_),"\n"; }
perl $agent->register('mylink', '.1.3.6.1.8765', \&myroutine);
```

This expression could also source an external file:

```
perl 'do /path/to/file.pl';
```

or perform any other perl-based processing that might be required.

### Dynamically Loadable Modules

Most of the MIBs supported by the Net-SNMP agent are implemented as C code modules, which were compiled and linked into the agent libraries when the suite was first built. Such implementation modules can also be compiled independently and loaded into the running agent once it has started. Use of this mechanism requires that the agent was built with support for the *ucd-snmp/dlmod* module (which is included as part of the default build configuration).

**dlmod NAME PATH**

will load the shared object module from the file PATH (an absolute filename), and call the initialisation routine *init\_NAME*.

Note: If the specified PATH is not a fully qualified filename, it will be interpreted relative to /code/git/mgt/target/install/lib/snmp/dlmod, and .so will be appended to the filename.

This functionality can also be configured using SNMP SET requests to the UCD-DLMOD-MIB.

**Proxy Support**

Another mechanism for extending the functionality of the agent is to pass selected requests (or selected varbinds) to another SNMP agent, which can be running on the same host (presumably listening on a different port), or on a remote system. This can be viewed either as the main agent delegating requests to the remote one, or acting as a proxy for it. Use of this mechanism requires that the agent was built with support for the *ucd-smnp/proxy* module (which is included as part of the default build configuration).

**proxy [-Cn CONTEXTNAME] [SNMPCMD\_ARGS] HOST OID [REMOTEOID]**

will pass any incoming requests under OID to the agent listening on the port specified by the transport address HOST. See the section **LISTENING ADDRESSES** in the *snmpd(8)* manual page for more information about the format of listening addresses.

Note: To proxy the entire MIB tree, use the OID .1.3 (**not** the top-level .1)

The *SNMPCMD\_ARGS* should provide sufficient version and administrative information to generate a valid SNMP request (see *snmpcmd(1)*).

Note: The proxied request will *not* use the administrative settings from the original request.

If a CONTEXTNAME is specified, this will register the proxy delegation within the named context in the local agent. Defining multiple *proxy* directives for the same OID but different contexts can be used to query several remote agents through a single proxy, by specifying the appropriate SNMPv3 context in the incoming request (or using suitable configured community strings - see the *com2sec* directive).

Specifying the REMOID parameter will map the local MIB tree rooted at OID to an equivalent subtree rooted at REMOID on the remote agent.

**SMUX Sub-Agents**

The Net-SNMP agent supports the SMUX protocol (RFC 1227) to communicate with SMUX-based sub-agents (such as *gated*, *zebra* or *quagga*). Use of this mechanism requires that the agent was built with support for the *smux* module, which is not part of the default build environment, and must be explicitly included by specifying the '--with-mib-modules=smux' option to the configure script when the package is first built.

Note: This extension protocol has been officially deprecated in favour of AgentX (see below).

**smuxpeer OID PASS**

will register a subtree for SMUX-based processing, to be authenticated using the password PASS. If a subagent (or "peer") connects to the agent and registers this subtree then requests for OIDs within it will be passed to that SMUX subagent for processing.

A suitable entry for an OSPF routing daemon (such as *gated*, *zebra* or *quagga*) might be something like

```
smuxpeer .1.3.6.1.2.1.14 ospf_pass
```

**smuxsocket <IPv4-address>**

defines the IPv4 address for SMUX peers to communicate with the Net-SNMP agent. The default is to listen on all IPv4 interfaces ("0.0.0.0"), unless the package has been configured with "--enable-local-smux" at build time, which causes it to only listen on 127.0.0.1 by default. SMUX uses the well-known TCP port 199.

Note the Net-SNMP agent will only operate as a SMUX *master* agent. It does not support acting in a SMUX subagent role.

### AgentX Sub-Agents

The Net-SNMP agent supports the AgentX protocol (RFC 2741) in both master and subagent roles. Use of this mechanism requires that the agent was built with support for the *agentx* module (which is included as part of the default build configuration), and also that this support is explicitly enabled (e.g. via the *snmpd.conf* file).

There are two directives specifically relevant to running as an AgentX master agent:

**master agentx**

will enable the AgentX functionality and cause the agent to start listening for incoming AgentX registrations. This can also be activated by specifying the '-x' command-line option (to specify an alternative listening socket).

**agentXPerms SOCKPERMS [DIRPERMS [USER|UID [GROUP|GID]]]**

Defines the permissions and ownership of the AgentX Unix Domain socket, and the parent directories of this socket. SOCKPERMS and DIRPERMS must be octal digits (see *chmod(1)*). By default this socket will only be accessible to subagents which have the same userid as the agent.

There is one directive specifically relevant to running as an AgentX sub-agent:

**agentXPingInterval NUM**

will make the subagent try and reconnect every NUM seconds to the master if it ever becomes (or starts) disconnected.

The remaining directives are relevant to both AgentX master and sub-agents:

**agentXSocket [<transport-specifier>:]<transport-address>[,...]**

defines the address the master agent listens at, or the subagent should connect to. The default is the Unix Domain socket `/var/agentx/master`. Another common alternative is `tcp:localhost:705`. See the section **LISTENING ADDRESSES** in the *snmpd(8)* manual page for more information about the format of addresses.

Note: Specifying an AgentX socket does **not** automatically enable AgentX functionality (unlike the '-x' command-line option).

**agentXTimeout NUM**

defines the timeout period (NUM seconds) for an AgentX request. Default is 1 second. NUM also be specified with a suffix of one of s (for seconds), m (for minutes), h (for hours), d (for days), or w (for weeks).

**agentXRetries NUM**

defines the number of retries for an AgentX request. Default is 5 retries.

net-snmp ships with both C and Perl APIs to develop your own AgentX subagent.

### OTHER CONFIGURATION

**override [-rw] OID TYPE VALUE**

This directive allows you to override a particular OID with a different value (and possibly a different type of value). The -rw flag will allow snmp SETs to modify it's value as well. (note that if you're overriding original functionality, that functionality will be entirely lost. Thus SETs will do nothing more than modify the internal overridden value and will not perform any of the original functionality intended to be provided by the MIB object. It's an emulation only.) An example:

```
override sysDescr.0 octet_str "my own sysDescr"
```

That line will set the sysDescr.0 value to "my own sysDescr" as well as make it modifiable with SNMP SETs as well (which is actually illegal according to the MIB specifications).

Note that care must be taken when using this. For example, if you try to override a property of the 3rd interface in the ifTable with a new value and later the numbering within the ifTable changes it's index ordering you'll end up with problems and your modified value won't appear in the right place in the table.

Valid TYPEs are: integer, uinteger, octet\_str, object\_id, counter, null (for gauges, use "uinteger"; for bit strings, use "octet\_str"). Note that setting an object to "null" effectively delete's it as being accessible. No VALUE needs to be given if the object type is null.

More types should be available in the future.

If you're trying to figure out aspects of the various mib modules (possibly some that you've added yourself), the following may help you spit out some useful debugging information. First off, please read the snmpd manual page on the -D flag. Then the following configuration snmpd.conf token, combined with the -D flag, can produce useful output:

injectHandler HANDLER modulename [beforeThis]

This will insert new handlers into the section of the mib tree referenced by "modulename". If "beforeThis" is specified then the module will be injected before the named module. This is useful for getting a handler into the exact right position in the chain.

The types of handlers available for insertion are:

stash\_cache

Caches information returned from the lower level. This greatly help the performance of the agent, at the cost of caching the data such that its no longer "live" for 30 seconds (in this future, this will be configurable). Note that this means snmpd will use more memory as well while the information is cached. Currently this only works for handlers registered using the table\_iterator support, which is only a few mib tables. To use it, you need to make sure to install it before the table\_iterator point in the chain, so to do this:

```
injectHandler stash_cache NAME table_iterator
```

If you want a table to play with, try walking the nsModuleTable with and without this injected.

debug Prints out lots of debugging information when the -Dhelper:debug flag is passed to the snmpd application.

read\_only

Forces turning off write support for the given module.

serialize

If a module is failing to handle multiple requests properly (using the new 5.0 module API), this will force the module to only receive one request at a time.

bulk\_to\_next

If a module registers to handle getbulk support, but for some reason is failing to implement it properly, this module will convert all getbulk requests to getnext requests before the final module receives it.

dontLogTCPWrappersConnects

If the **snmpd** was compiled with TCP Wrapper support, it logs every connection made to the agent. This setting disables the log messages for accepted connections. Denied connections will still be logged.

Figuring out module names

To figure out which modules you can inject things into, run **snmpwalk** on the nsModuleTable which will give a list of all named modules registered within the agent.

### Internal Data tables

table NAME

add\_row NAME INDEX(ES) VALUE(S)

## NOTES

- o The Net-SNMP agent can be instructed to re-read the various configuration files, either via an **snmpset** assignment of integer(1) to UCD-SNMP-MIB::versionUpdateConfig.0 (.1.3.6.1.4.1.2021.100.11.0), or by sending a **kill -HUP** signal to the agent process.
- o All directives listed with a value of "yes" actually accept a range of boolean values. These will accept any of *1*, *yes* or *true* to enable the corresponding behaviour, or any of *0*, *no* or *false* to disable it. The default in each case is for the feature to be turned off, so these directives are typically only used to enable the appropriate behaviour.

## EXAMPLE CONFIGURATION FILE

See the EXAMPLE.CONF file in the top level source directory for a more detailed example of how the above information is used in real examples.

## FILES

/code/git/mgt/target/install/etc/snmp/snmpd.conf

## SEE ALSO

snmpconf(1), snmpusm(1), snmp.conf(5), snmp\_config(5), snmpd(8), EXAMPLE.conf, netsnmp\_config\_api(3).