Standard Documentation for Net::SNMP, Net::LDAP and Perl Data Structures

Contents

Chapter 1

Net::SNMP

1.1 NAME

Net::SNMP - Object oriented interface to SNMP

1.2 SYNOPSIS

The Net::SNMP module implements an object oriented interface to the Simple Network Management Protocol. Perl applications can use the module to retrieve or update information on a remote host using the SNMP protocol. The module supports SNMP version-1, SNMP version-2c (Community-Based SNMPv2), and SNMP version-3. The Net::SNMP module assumes that the user has a basic understanding of the Simple Network Management Protocol and related network management concepts.

1.3 DESCRIPTION

The Net::SNMP module abstracts the intricate details of the Simple Network Management Protocol by providing a high level programming interface to the protocol. Each Net::SNMP object provides a one-to-one mapping between a Perl object and a remote SNMP agent or manager. Once an object is created, it can be used to perform the basic protocol exchange actions defined by SNMP.

A Net::SNMP object can be created such that it has either "blocking" or "non-blocking" properties. By default, the methods used to send SNMP messages do not return until the protocol exchange has completed successfully or a timeout period has expired. This behavior gives the object a "blocking" property because the flow of the code is stopped until the method returns.

The optional named argument **-nonblocking** can be passed to the object constructor with a true value to give the object "non-blocking" behavior. A method invoked by a non-blocking object queues the SNMP message and returns immediately, allowing the flow of the code to continue. The queued SNMP messages are not sent until an event loop is entered by calling the `snmp_dispatcher()` method. When the SNMP messages are sent, any response to the messages invokes the subroutine defined by the user when the message was originally queued. The event loop exits when all messages have been removed from the queue by either receiving a response, or by exceeding the number of retries at the Transport Layer.

Blocking Objects

The default behavior of the methods associated with a Net::SNMP object is to block the code flow until the method completes. For methods that initiate a SNMP protocol exchange requiring a response, a hash reference containing the results of the query is returned. The undefined value is returned by all methods when a failure has occurred. The `error()` method can be used to determine the cause of the failure.

The hash reference returned by a SNMP protocol exchange points to a hash constructed from the VarBindList contained in the SNMP response message. The hash is created using the ObjectName and the ObjectSyntax pairs in the VarBindList. The keys of the hash consist of the OBJECT IDENTIFIERs in dotted notation corresponding to each ObjectName in the VarBindList. The value of each hash entry is set equal to the value of the corresponding ObjectSyntax. This hash reference can also be retrieved using the `var_bind_list()` method.

Non-blocking Objects

When a Net::SNMP object is created having non-blocking behavior, the invocation of a method associated with the object returns immediately, allowing the flow of the code to continue. When a method is invoked that would initiate a SNMP protocol exchange requiring a response, either a true value (i.e. 0x1) is returned immediately or the undefined value is returned if there was a failure. The `error()` method can be used to determine the cause of the failure.

The contents of the VarBindList contained in the SNMP response message can be retrieved by calling the `var_bind_list()` method using the object reference passed as the first argument to the callback. The value returned by the `var_bind_list()` method is a hash reference created using the ObjectName and the ObjectSyntax pairs in the VarBindList. The keys of the hash consist of the OBJECT IDENTIFIERs in dotted notation corresponding to each ObjectName in the VarBindList. The value of each hash entry is set equal to the value of the corresponding ObjectSyntax. The undefined value is returned if there has been a failure and the `error()` method may be used to determine the reason.

1.4 METHODS

When named arguments are expected by the methods, two different styles are supported. All examples in this documentation use the dashed-option style:

```
$object->method(-argument => $value);
```

However, the IO:: style is also allowed:

```
$object->method(Argument => $value);
```

Non-blocking Objects Arguments

When a Net::SNMP object has been created with a "non-blocking" property, most methods that generate a SNMP message take additional arguments to support this property.

Callback

Most methods associated with a non-blocking object have an optional named argument called **-callback**. The **-callback** argument expects a reference to a subroutine or to an array whose first element must be a reference to a subroutine. The subroutine defined by the **-callback** option is executed when a response to a SNMP message is received, an error condition has occurred, or the number of retries for the message has been exceeded.

When the **-callback** argument only contains a subroutine reference, the subroutine is evaluated passing a reference to the original Net::SNMP object as the only parameter. If the **-callback** argument was defined as an array reference, all elements in the array are passed to subroutine after the reference to the Net::SNMP object. The first element, which is required to be a reference to a subroutine, is removed before the remaining arguments are passed to that subroutine.

Once one method is invoked with the **-callback** argument, this argument stays with the object and is used by any further calls to methods using the **-callback** option if the argument is absent. The undefined value may be passed to the **-callback** argument to delete the callback.

NOTE: The subroutine being passed with the **-callback** named argument should not cause blocking itself. This will cause all the actions in the event loop to be stopped, defeating the non-blocking property of the Net::SNMP module.

Delay

An optional argument **-delay** can also be passed to non-blocking objects. The **-delay** argument instructs the object to wait the number of seconds passed to the argument before executing the SNMP protocol exchange. The delay period starts when the event loop is entered. The **-delay** parameter is applied to all methods associated with the object once it is specified. The delay value must be set back to 0 seconds to disable the delay parameter.

SNMPv3 Arguments

A SNMP context is a collection of management information accessible by a SNMP entity. An item of management information may exist in more than one context and a SNMP entity potentially has access to many contexts. The combination of a contextEngineID and a contextName unambiguously identifies a context within an administrative domain. In a SNMPv3 message, the contextEngineID and contextName are included as part of the scopedPDU. All methods that generate a SNMP message optionally take a **-contextengineid** and **-contextname** argument to configure these fields.

Context Engine ID

The **-contextengineid** argument expects a hexadecimal string representing the desired contextEngineID. The string must be 10 to 64 characters (5 to 32 octets) long and can be prefixed with an optional "0x". Once the **-contextengineid** is specified it stays with the object until it is changed again or reset to default by passing in the undefined value. By default, the contextEngineID is set to match the authoritativeEngineID of the authoritative SNMP engine.

Context Name

The contextName is passed as a string which must be 0 to 32 octets in length using the **-contextname** argument. The contextName stays with the object until it is changed. The contextName defaults to an empty string which represents the "default" context.

session() - create a new Net::SNMP object

```
($session, $error) = Net::SNMP->session(
                        [-hostname      => $hostname,]
                        [-port          => $port,]
                        [-localaddr     => $localaddr,]
                        [-localport     => $localport,]
                        [-nonblocking   => $boolean,]
                        [-version       => $version,]
                        [-timeout       => $seconds,]
                        [-retries       => $count,]
                        [-maxmsgsize    => $octets,]
                        [-translate     => $translate,]
                        [-debug         => $bitmask,]
                        [-community     => $community,]   # v1/v2c
                        [-username      => $username,]    # v3
                        [-authkey       => $authkey,]     # v3
                        [-authpassword  => $authpasswd,]  # v3
                        [-authprotocol  => $authproto,]   # v3
                        [-privkey       => $privkey,]     # v3
                        [-privpassword  => $privpasswd,]  # v3
                        [-privprotocol  => $privproto,]   # v3
                     );
```

This is the constructor for Net::SNMP objects. In scalar context, a reference to a new Net::SNMP object is returned if the creation of the object is successful. In list context, a reference to a new Net::SNMP object and an empty error message string is returned. If a failure occurs, the object reference is returned as the undefined value. The error string may be used to determine the cause of the error.

Most of the named arguments passed to the constructor define basic attributes for the object and are not modifiable after the object has been created. The **-timeout**, **-retries**, **-maxmsgsize**, **-translate**, and **-debug** arguments are modifiable using an accessor method. See their corresponding method definitions for a complete description of their usage, default values, and valid ranges.

Transport Layer Arguments

The Net::SNMP module uses UDP/IP as the Transport Layer to pass SNMP messages between the local and remote devices. The destination device can be specified using the **-hostname** argument. The **-hostname** argument accepts either an IP network hostname or an IP address in dotted notation. This argument is optional and defaults to "localhost". The destination UPD port number can be specified using the **-port** argument. This argument is also optional and defaults to 161, which is the port number on which devices using default values expect to receive SNMP request messages. The **-port** argument will need to be specified for remote devices expecting to receive SNMP notifications since these device typically default to port 162.

By default, the source IP address and port number are assigned dynamically by the local device on which the Net::SNMP module is being used. This dynamic assignment can be overridden by using the **-localaddr** and **-localport** arguments. These values default to *INADDR_ANY* (typically 0.0.0.0) and 0 respectively. The **-localaddr** argument will accept either an IP network hostname or an IP address in dotted notation. If a hostname is specified, the resolved IP address must be a valid address on the local device.

Security Model Arguments

The **-version** argument controls which other arguments are expected or required by the `session()` constructor. The Net::SNMP module supports SNMPv1, SNMPv2c, and SNMPv3. The module defaults to SNMPv1 if no **-version** argument is specified. The **-version** argument expects either a digit (i.e. '1', '2', or '3') or a string specifying the version (i.e. 'snmpv1', 'snmpv2c', or 'snmpv3') to define the SNMP version.

The Security Model used by the Net::SNMP object is based on the SNMP version associated with the object. If the SNMP version is SNMPv1 or SNMPv2c a Community-based Security Model will be used, while the User-based Security Model (USM) will be used if the version is SNMPv3.

Community-based Security Model Argument

If the Security Model is Community-based, the only argument available is the **-community** argument. This argument expects a string that is to be used as the SNMP community name. By default the community name is set to 'public' if the argument is not present.

User-based Security Model Arguments

The User-based Security Model (USM) used by SNMPv3 requires that a securityName be specified using the **-username** argument. The creation of a Net::SNMP object with the version set to SNMPv3 will fail if the **-username** argument is not present. The **-username** argument expects a string 1 to 32 octets in length.

Different levels of security are allowed by the User-based Security Model which address authentication and privacy concerns. A SNMPv3 Net::SNMP object will derive the security level (securityLevel) based on which of the following arguments are specified.

By default a securityLevel of 'noAuthNoPriv' is assumed. If the **-authkey** or **-authpassword** arguments are specified, the securityLevel becomes 'authNoPriv'. The **-authpassword** argument expects a string which is at least 1 octet in length. Optionally, the **-authkey** argument can be used so that a plain text password does not have to be specified in a script. The **-authkey** argument expects a hexadecimal string produced by localizing the password with the authoritativeEngineID for the specific destination device. The `snmpkey` utility included with the distribution can be used to create the hexadecimal string (see *snmpkey*).

Two different hash algorithms are defined by SNMPv3 which can be used by the Security Model for authentication. These algorithms are HMAC-MD5-96 "MD5" (RFC 1321) and HMAC-SHA-96 "SHA-1" (NIST FIPS PUB 180-1). The default algorithm used by the module is HMAC-MD5-96. This behavior can be changed by using the **-authprotocol** argument. This argument expects either the string 'md5' or 'sha' to be passed to modify the hash algorithm.

By specifying the arguments **-privkey** or **-privpassword** the securityLevel associated with the object becomes 'authPriv'. According to SNMPv3, privacy requires the use of authentication. Therefore, if either of these two arguments are present and the **-authkey** or **-authpassword** arguments are missing, the creation of the object fails. The **-privkey** and **-privpassword** arguments expect the same input as the **-authkey** and **-authpassword** arguments respectively.

The User-based Security Model described in RFC 3414 defines a single encryption protocol to be used for privacy. This protocol, CBC-DES "DES" (NIST FIPS PUB 46-1), is used by default or if the string 'des' is passed to the **-privprotocol** argument. By working with the Extended Security Options Consortium http://www.snmp.com/eso/, the module also supports additional protocols which have been defined in draft specifications. The draft http://www.snmp.com/eso/draft-reeder-snmpv3-usm-3desede-00.txt defines the support of CBC-3DES-EDE "Triple-DES" (NIST FIPS 46-3) in the User-based Security Model. This protocol can be selected using the **-privprotocol** argument with the string '3desede'. The draft http://www.snmp.com/eso/draft-blumenthal-aes-usm-04.txt describes the use of CFB128-AES-128/192/256 "AES" (NIST FIPS PUB 197) in the USM. The three AES encryption protocols, differentiated by their key sizes, can be selected by passing 'aescfb128', 'aescfb192', or 'aescfb256' to the **-privprotocol** argument.

close() - clear the Transport Layer associated with the object

```
$session->close;
```

This method clears the UDP Transport Layer and any errors associated with the object. Once closed, the Net::SNMP object can no longer be used to send or receive SNMP messages.

snmp_dispatcher() - enter the non-blocking object event loop

```
$session->snmp_dispatcher();
```

This method enters the event loop associated with non-blocking Net::SNMP objects. The method exits when all queued SNMP messages have received a response or have timed out at the Transport Layer. This method is also exported as the stand alone function `snmp_dispatcher()` by default (see §1.6).

get_request() - send a SNMP get-request to the remote agent

```
$result = $session->get_request(
                        [-callback        => sub {},]     # non-blocking
                        [-delay           => $seconds,]   # non-blocking
                        [-contextengineid => $engine_id,] # v3
                        [-contextname     => $name,]      # v3
                        -varbindlist      => \@oids,
                      );
```

This method performs a SNMP get-request query to gather data from the remote agent on the host associated with the Net::SNMP object. The message is built using the list of OBJECT IDENTIFIERs in dotted notation passed to the method as an array reference using the **-varbindlist** argument. Each OBJECT IDENTIFER is placed into a single SNMP GetRequest-PDU in the same order that it held in the original list.

A reference to a hash is returned in blocking mode which contains the contents of the VarBindList. In non-blocking mode, a true value is returned when no error has occurred. In either mode, the undefined value is returned when an error has occurred. The `error()` method may be used to determine the cause of the failure.

get_next_request() - send a SNMP get-next-request to the remote agent

```
$result = $session->get_next_request(
                        [-callback        => sub {},]     # non-blocking
                        [-delay           => $seconds,]   # non-blocking
                        [-contextengineid => $engine_id,] # v3
                        [-contextname     => $name,]      # v3
                        -varbindlist      => \@oids,
                      );
```

This method performs a SNMP get-next-request query to gather data from the remote agent on the host associated with the Net::SNMP object. The message is built using the list of OBJECT IDENTIFIERs in dotted notation passed to the method as an array reference using the **-varbindlist** argument. Each OBJECT IDENTIFER is placed into a single SNMP GetNextRequest-PDU in the same order that it held in the original list.

A reference to a hash is returned in blocking mode which contains the contents of the VarBindList. In non-blocking mode, a true value is returned when no error has occurred. In either mode, the undefined value is returned when an error has occurred. The `error()` method may be used to determine the cause of the failure.

set_request() - send a SNMP set-request to the remote agent

```
$result = $session->set_request(
                        [-callback        => sub {},]     # non-blocking
                        [-delay           => $seconds,]   # non-blocking
                        [-contextengineid => $engine_id,] # v3
                        [-contextname     => $name,]      # v3
                        -varbindlist      => \@oid_value,
                      );
```

This method is used to modify data on the remote agent that is associated with the Net::SNMP object using a SNMP set-request. The message is built using a list of values consisting of groups of an OBJECT IDENTIFIER, an object type, and the actual value to be set. This list is passed to the method as an array reference using the **-varbindlist** argument. The OBJECT IDENTIFIERs in each trio are to be in dotted

notation. The object type is an octet corresponding to the ASN.1 type of value that is to be set. Each of the supported ASN.1 types have been defined and are exported by the package by default (see §1.6).

A reference to a hash is returned in blocking mode which contains the contents of the VarBindList. In non-blocking mode, a true value is returned when no error has occurred. In either mode, the undefined value is returned when an error has occurred. The `error()` method may be used to determine the cause of the failure.

trap() - send a SNMP trap to the remote manager

```
$result = $session->trap(
                        [-delay           => $seconds,]   # non-blocking
                        [-enterprise      => $oid,]
                        [-agentaddr       => $ipaddress,]
                        [-generictrap     => $generic,]
                        [-specifictrap    => $specific,]
                        [-timestamp       => $timeticks,]
                        -varbindlist      => \@oid_value,
                      );
```

This method sends a SNMP trap to the remote manager associated with the Net::SNMP object. All arguments are optional and will be given the following defaults in the absence of a corresponding named argument:

- The default value for the trap **-enterprise** is "1.3.6.1.4.1", which corresponds to "`iso.org.dod.internet.private.enterprises`". The enterprise value is expected to be an OBJECT IDENTIFER in dotted notation.
- The default value for the trap **-agentaddr** is the local IP address from the host on which the script is running or the local address specified by the **-localaddr** option. The agent-addr is expected to be an IpAddress in dotted notation.
- The default value for the **-generictrap** type is 6 which corresponds to "enterpriseSpecific". The generic-trap types are defined and can be exported upon request (see §1.6).
- The default value for the **-specifictrap** type is 0. No pre-defined values are available for specific-trap types.
- The default value for the trap **-timestamp** is the "uptime" of the script. The "uptime" of the script is the number of hundredths of seconds that have elapsed since the script began running. The time-stamp is expected to be a TimeTicks number in hundredths of seconds.
- The default value for the trap **-varbindlist** is an empty array reference. The variable-bindings are expected to be in an array format consisting of groups of an OBJECT IDENTIFIER, an object type, and the actual value of the object. This is identical to the list expected by the `set_request()` method. The OBJECT IDENTIFIERs in each trio are to be in dotted notation. The object type is an octet corresponding to the ASN.1 type for the value. Each of the supported types have been defined and are exported by default (see §1.6).

A true value is returned when the method is successful. The undefined value is returned when a failure has occurred. The `error()` method can be used to determine the cause of the failure. Since there are no acknowledgements for Trap-PDUs, there is no way to determine if the remote host actually received the trap.

NOTE: When the object is in non-blocking mode, the trap is not sent until the event loop is entered and no callback is ever executed.

NOTE: This method can only be used when the version of the object is set to SNMPv1.

get_bulk_request() - send a get-bulk-request to the remote agent

```
$result = $session->get_bulk_request(
                        [-callback        => sub {},]     # non-blocking
                        [-delay           => $seconds,]   # non-blocking
                        [-contextengineid => $engine_id,] # v3
                        [-contextname     => $name,]      # v3
```

```
                        [-nonrepeaters    => $non_reps,]
                        [-maxrepetitions  => $max_reps,]
                        -varbindlist      => \@oids,
                    );
```

This method performs a SNMP get-bulk-request query to gather data from the remote agent on the host associated with the Net::SNMP object. All arguments are optional except **-varbindlist** and will be given the following defaults in the absence of a corresponding named argument:

- The default value for the get-bulk-request **-nonrepeaters** is 0. The non-repeaters value specifies the number of variables in the variable-bindings list for which a single successor is to be returned.
- The default value for the get-bulk-request **-maxrepetitions** is 0. The max-repetitions value specifies the number of successors to be returned for the remaining variables in the variable-bindings list.
- The **-varbindlist** argument expects an array reference consisting of a list of OBJECT IDENTIFIERs in dotted notation. Each OBJECT IDENTIFER is placed into a single SNMP GetBulkRequest-PDU in the same order that it held in the original list.

A reference to a hash is returned in blocking mode which contains the contents of the VarBindList. In non-blocking mode, a true value is returned when no error has occurred. In either mode, the undefined value is returned when an error has occurred. The `error()` method may be used to determine the cause of the failure.

NOTE: This method can only be used when the version of the object is set to SNMPv2c or SNMPv3.

inform_request() - send an inform-request to the remote manager

```
$result = $session->inform_request(
                        [-callback        => sub {},]     # non-blocking
                        [-delay           => $seconds,]   # non-blocking
                        [-contextengineid => $engine_id,] # v3
                        [-contextname     => $name,]      # v3
                        -varbindlist      => \@oid_value,
                    );
```

This method is used to provide management information to the remote manager associated with the Net::SNMP object using an inform-request. The message is built using a list of values consisting of groups of an OBJECT IDENTIFIER, an object type, and the actual value to be identified. This list is passed to the method as an array reference using the **-varbindlist** argument. The OBJECT IDENTIFIERs in each trio are to be in dotted notation. The object type is an octet corresponding to the ASN.1 type of value that is to be identified. Each of the supported ASN.1 types have been defined and are exported by the package by default (see §1.6).

The first two variable-bindings fields in the inform-request are specified by SNMPv2 and should be:

- sysUpTime.0 - ('1.3.6.1.2.1.1.3.0', TIMETICKS, $timeticks)
- snmpTrapOID.0 - ('1.3.6.1.6.3.1.1.4.1.0', OBJECT_IDENTIFIER, $oid)

A reference to a hash is returned in blocking mode which contains the contents of the VarBindList. In non-blocking mode, a true value is returned when no error has occurred. In either mode, the undefined value is returned when an error has occurred. The `error()` method may be used to determine the cause of the failure.

NOTE: This method can only be used when the version of the object is set to SNMPv2c or SNMPv3.

snmpv2_trap() - send a snmpV2-trap to the remote manager

```
$result = $session->snmpv2_trap(
                        [-delay           => $seconds,]   # non-blocking
                        -varbindlist      => \@oid_value,
                    );
```

This method sends a snmpV2-trap to the remote manager associated with the Net::SNMP object. The message is built using a list of values consisting of groups of an OBJECT IDENTIFIER, an object type, and the actual value to be identified. This list is passed to the method as an array reference using the **-varbindlist** argument. The OBJECT IDENTIFIERs in each trio are to be in dotted notation. The object type is an octet corresponding to the ASN.1 type of value that is to be identified. Each of the supported ASN.1 types have been defined and are exported by the package by default (see §1.6).

The first two variable-bindings fields in the snmpV2-trap are specified by SNMPv2 and should be:

- sysUpTime.0 - ('1.3.6.1.2.1.1.3.0', TIMETICKS, $timeticks)
- snmpTrapOID.0 - ('1.3.6.1.6.3.1.1.4.1.0', OBJECT_IDENTIFIER, $oid)

A true value is returned when the method is successful. The undefined value is returned when a failure has occurred. The `error()` method can be used to determine the cause of the failure. Since there are no acknowledgements for SNMPv2-Trap-PDUs, there is no way to determine if the remote host actually received the snmpV2-trap.

NOTE: When the object is in non-blocking mode, the snmpV2-trap is not sent until the event loop is entered and no callback is ever executed.

NOTE: This method can only be used when the version of the object is set to SNMPv2c. SNMPv2-Trap-PDUs are supported by SNMPv3, but require the sender of the message to be an authoritative SNMP engine which is not currently supported by the Net::SNMP module.

get_table() - retrieve a table from the remote agent

```
$result = $session->get_table(
                        [-callback        => sub {},]     # non-blocking
                        [-delay           => $seconds,]   # non-blocking
                        [-contextengineid => $engine_id,] # v3
                        [-contextname     => $name,]      # v3
                        -baseoid          => $oid,
                        [-maxrepetitions  => $max_reps,]  # v2c/v3
                      );
```

This method performs repeated SNMP get-next-request or get-bulk-request (when using SNMPv2c or SNMPv3) queries to gather data from the remote agent on the host associated with the Net::SNMP object. The first message sent is built using the OBJECT IDENTIFIER in dotted notation passed to the method by the **-baseoid** argument. Repeated SNMP requests are issued until the OBJECT IDENTIFER in the response is no longer a child of the base OBJECT IDENTIFIER.

The **-maxrepetitions** argument can be used to specify the max-repetitions value that is passed to the get-bulk-requests when using SNMPv2c or SNMPv3. If this argument is not present, a value is calculated based on the maximum message size for the Net::SNMP object.

A reference to a hash is returned in blocking mode which contains the contents of the VarBindList. In non-blocking mode, a true value is returned when no error has occurred. In either mode, the undefined value is returned when an error has occurred. The `error()` method may be used to determine the cause of the failure.

WARNING: Results from this method can become very large if the base OBJECT IDENTIFIER is close to the root of the SNMP MIB tree.

get_entries() - retrieve table entries from the remote agent

```
$result = $session->get_entries(
                        [-callback        => sub {},]     # non-blocking
                        [-delay           => $seconds,]   # non-blocking
                        [-contextengineid => $engine_id,] # v3
                        [-contextname     => $name,]      # v3
                        -columns          => \@columns,
                        [-startindex      => $start,]
                        [-endindex        => $end,]
                        [-maxrepetitions  => $max_reps,]  # v2c/v3
                      );
```

This method performs repeated SNMP get-next-request or get-bulk-request (when using SNMPv2c or SNMPv3) queries to gather data from the remote agent on the host associated with the Net::SNMP object. Each message specifically requests data for each OBJECT IDENTIFIER specified in the **-columns** array. The OBJECT IDENTIFIERs must correspond to column entries for a conceptual row in a table. They may however be columns in different tables as long as each table is indexed the same way. The optional **-startindex** and **-endindex** arguments may be specified to limit the query to specific rows in the table(s).

The **-startindex** can be specified as a single decimal value or in dotted notation if the index associated with the entry so requires. If the **-startindex** is specified, it will be include as part of the query results. If no **-startindex** is specified, the first request message will be sent without an index. To insure that the **-startindex** is included, the last subidentifier in the index is decremented by one. If the last subidentifier has a value of zero, the subidentifier is removed from the index.

The optional **-endindex** argument can be specified as a single decimal value or in dotted notation. If the **-endindex** is specified, it will be included as part of the query results. If no **-endindex** is specified, repeated SNMP requests are issued until the response no longer returns entries matching any of the columns specified in the **-columns** array.

The **-maxrepetitions** argument can be used to specify the max-repetitions value that is passed to the get-bulk-requests when using SNMPv2c or SNMPv3. If this argument is not present, a value is calculated based on the maximum message size of the object and the number of columns specified in the **-columns** array.

A reference to a hash is returned in blocking mode which contains the contents of the VarBindList. In non-blocking mode, a true value is returned when no error has occurred. In either mode, the undefined value is returned when an error has occurred. The `error()` method may be used to determine the cause of the failure.

version() - get the SNMP version from the object

```
$rfc_version = $session->version;
```

This method returns the current value for the SNMP version associated with the object. The returned value is the corresponding version number defined by the RFCs for the protocol version field (i.e. SNMPv1 == 0, SNMPv2c == 1, and SNMPv3 == 3). The RFC versions are defined as constant by the module and can be exported by request (see §1.6).

error() - get the current error message from the object

```
$error_message = $session->error;
```

This method returns a text string explaining the reason for the last error. An empty string is returned if no error has occurred.

hostname() - get the hostname associated with the object

```
$hostname = $session->hostname;
```

This method returns the hostname string that is associated with the object as it was passed to the `session()` constructor.

error_status() - get the current SNMP error-status from the object

```
$error_status = $session->error_status;
```

This method returns the numeric value of the error-status contained in the last SNMP GetResponse-PDU received by the object.

error_index() - get the current SNMP error-index from the object

```
$error_index = $session->error_index;
```

This method returns the numeric value of the error-index contained in the last SNMP GetResponse-PDU received by the object.

var_bind_list() - get the hash reference to the last SNMP response

```
$response = $session->var_bind_list;
```

This method returns a hash reference created using the ObjectName and the ObjectSyntax pairs in the VarBindList of the last SNMP GetResponse-PDU received by the object. The keys of the hash consist of the OBJECT IDENTIFIERs in dotted notation corresponding to each ObjectName in the VarBindList. If any of the OBJECT IDENTIFIERs passed to the request method began with a leading dot, all of the OBJECT IDENTIFIER hash keys will be prefixed with a leading dot. The value of each hash entry is set equal to the value of the corresponding ObjectSyntax. The undefined value is returned if there has been a failure and the `error()` method may be used to determine the reason.

var_bind_names() - get the array of the ObjectNames in the last response

```
@names = $session->var_bind_names;
```

This method returns an array containing the OBJECT IDENTIFIERs corresponding to the ObjectNames in the VarBindList in the order that they were received in the last SNMP GetResponse-PDU. The entries in the array will map directly to the keys in the hash reference returned by the methods that perform SNMP message exchanges and by the `var_bind_list()` method. The array returned for the convenience methods `get_table()` and `get_entries()` will be in lexicographical order. An empty array is returned if there has been a failure and the `error()` method may be used to determine reason.

timeout() - set or get the current timeout period for the object

```
$seconds = $session->timeout([$seconds]);
```

This method returns the current value for the Transport Layer timeout for the Net::SNMP object. This value is the number of seconds that the object will wait for a response from the agent on the remote host. The default timeout is 5.0 seconds.

If a parameter is specified, the timeout for the object is set to the provided value if it falls within the range 1.0 to 60.0 seconds. The undefined value is returned upon an error and the `error()` method may be used to determine the cause.

retries() - set or get the current retry count for the object

```
$count = $session->retries([$count]);
```

This method returns the current value for the number of times to retry sending a SNMP message to the remote host. The default number of retries is 1.

If a parameter is specified, the number of retries for the object is set to the provided value if it falls within the range 0 to 20. The undefined value is returned upon an error and the `error()` method may be used to determine the cause.

max_msg_size() - set or get the current maxMsgSize for the object

```
$octets = $session->max_msg_size([$octets]);
```

This method returns the current value for the maximum message size (maxMsgSize) for the Net::SNMP object. This value is the largest message size in octets that can be prepared or processed by the object. The default maxMsgSize is 1472 octets.

If a parameter is specified, the maxMsgSize is set to the provided value if it falls within the range 484 to 2147483647 octets. The undefined value is returned upon an error and the `error()` method may be used to determine the cause.

NOTE: When using SNMPv3, the maxMsgSize is actually contained in the SNMP message (as msgMaxSize). If the value received from a remote device is less than the current maxMsgSize, the size is automatically adjusted to be the lower value.

translate() - enable or disable the translation mode for the object

```
$mask = $session->translate([
                    $mode |
                    [ # Perl anonymous ARRAY reference
                       ['-all'            => $mode0,]
                       ['-none'           => $mode1,]
                       ['-octetstring     => $mode2,]
                       ['-null'           => $mode3,]
                       ['-timeticks'      => $mode4,]
                       ['-opaque'         => $mode5,]
                       ['-nosuchobject'   => $mode6,]
                       ['-nosuchinstance' => $mode7,]
                       ['-endofmibview'   => $mode8,]
                       ['-unsigned'       => $mode9]
                    ]
                ]);
```

When the object decodes the GetResponse-PDU that is returned in response to a SNMP message, certain values are translated into a more "human readable" form. By default the following translations occur:

- OCTET STRINGs and Opaques containing non-printable ASCII characters are converted into a hexadecimal representation prefixed with "0x". **NOTE:** The following ASCII control characters are considered to be printable by the module: NUL(0x00), HT(0x09), LF(0x0A), FF(0x0C), and CR(0x0D).
- TimeTicks integer values are converted to a time format.
- NULL values return the string "NULL" instead of an empty string.
- noSuchObject exception values return the string "noSuchObject" instead of an empty string. If translation is not enabled, the SNMP error-status field is set to 128 which is equal to the exported definition NOSUCHOBJECT (see §1.6).
- noSuchInstance exception values return the string "noSuchInstance" instead of an empty string. If translation is not enabled, the SNMP error-status field is set to 129 which is equal to the exported definition NOSUCHINSTANCE (see §1.6).
- endOfMibView exception values return the string "endOfMibView" instead of an empty string. If translation is not enabled, the SNMP error-status field is set to 130 which is equal to the exported definition ENDOFMIBVIEW (see §1.6).
- Counter64, Counter, Gauge, and TimeTick values that have been incorrectly encoded as signed negative values are returned as unsigned values.

The `translate()` method can be invoked with two different types of arguments.

If the argument passed is any Perl variable type except an array reference, the translation mode for all ASN.1 types is set to either enabled or disabled, depending on the value of the passed parameter. Any value that Perl would treat as a true value will set the mode to be enabled for all types, while a false value will disable translation for all types.

A reference to an array can be passed to the `translate()` method in order to define the translation mode on a per ASN.1 type basis. The array is expected to contain a list of named argument pairs for each ASN.1 type that is to be modified. The arguments in the list are applied in the order that they are passed in via the array. Arguments at the end of the list supercede those passed earlier in the list. The argument "-all" can be used to specify that the mode is to apply to all ASN.1 types. Only the arguments for the ASN.1 types that are to be modified need to be included in the list.

The `translate()` method returns a bit mask indicating which ASN.1 types are to be translated. Definitions of the bit to ASN.1 type mappings can be exported using the *:translate* tag (see §1.6). The undefined value is returned upon an error and the `error()` method may be used to determine the cause.

debug() - set or get the debug mode for the module

```
$mask = $session->debug([$mask]);
```

This method is used to enable or disable debugging for the Net::SNMP module. Debugging can be enabled on a per component level as defined by a bit mask passed to the `debug()` method. The bit mask is broken up as follows:

- 0x02 - Message or PDU encoding and decoding
- 0x04 - Transport Layer
- 0x08 - Dispatcher
- 0x10 - Message Processing
- 0x20 - Security

Symbols representing these bit mask values are defined by the module and can be exported using the *:debug* tag (see §1.6). If a non-numeric value is passed to the `debug()` method, it is evaluated in boolean context. Debugging for all of the components is then enabled or disabled based on the resulting truth value.

The current debugging mask is returned by the method. Debugging can also be enabled using the stand alone function `snmp_debug()`. This function can be exported by request (see §1.6).

1.5 FUNCTIONS

oid_base_match() - determine if an OID has a specified OID base

```
$value = oid_base_match($base_oid, $oid);
```

This function takes two OBJECT IDENTIFIERs in dotted notation and returns a true value (i.e. 0x1) if the second OBJECT IDENTIFIER is equal to or is a child of the first OBJECT IDENTIFIER in the SNMP Management Information Base (MIB). This function can be used in conjunction with the `get-next-request()` or `get-bulk-request()` methods to determine when a OBJECT IDENTIFIER in the GetResponse-PDU is no longer in the desired MIB tree branch.

oid_lex_sort() - sort a list of OBJECT IDENTIFIERs lexicographically

```
@sorted_oids = oid_lex_sort(@oids);
```

This function takes a list of OBJECT IDENTIFIERs in dotted notation and returns the listed sorted in lexicographical order.

ticks_to_time() - convert TimeTicks to formatted time

```
$time = ticks_to_time($timeticks);
```

This function takes an ASN.1 TimeTicks value and returns a string representing the time defined by the value. The TimeTicks value is expected to be a non-negative integer value representing the time in hundredths of a second since some epoch. The returned string will display the time in days, hours, and seconds format according to the value of the TimeTicks argument.

1.6 EXPORTS

The Net::SNMP module uses the *Exporter* module to export useful constants and subroutines. These exportable symbols are defined below and follow the rules and conventions of the *Exporter* module (see *Exporter*).

Default

&snmp_dispatcher, INTEGER, INTEGER32, OCTET_STRING, OBJECT_IDENTIFIER, IPADDRESS, COUNTER, COUNTER32, GAUGE, GAUGE32, UNSIGNED32, TIMETICKS, OPAQUE, COUNTER64, NOSUCHOBJECT, NOSUCHINSTANCE, ENDOFMIBVIEW

Exportable

&snmp_debug, &snmp_dispatcher, &oid_base_match, &oid_lex_sort, &ticks_to_time, INTEGER, INTEGER32, OCTET_STRING, NULL, OBJECT_IDENTIFIER, SEQUENCE, IPADDRESS, COUNTER, COUNTER32, GAUGE, GAUGE32, UNSIGNED32, TIMETICKS, OPAQUE, COUNTER64, NOSUCHOBJECT, NOSUCHINSTANCE, ENDOFMIBVIEW, GET_REQUEST, GET_NEXT_REQUEST, GET_RESPONSE, SET_REQUEST, TRAP, GET_BULK_REQUEST, INFORM_REQUEST, SNMPV2_TRAP, DEBUG_ALL, DEBUG_NONE, DEBUG_MESSAGE, DEBUG_TRANSPORT, DEBUG_DISPATCHER, DEBUG_PROCESSING, DEBUG_SECURITY, COLD_START, WARM_START, LINK_DOWN, LINK_UP, AUTHENTICATION_FAILURE, EGP_NEIGHBOR_LOSS, ENTERPRISE_SPECIFIC, SNMP_VERSION_1, SNMP_VERSION_2C, SNMP_VERSION_3, SNMP_PORT, SNMP_TRAP_PORT, TRANSLATE_NONE, TRANSLATE_OCTET_STRING, TRANSLATE_NULL, TRANSLATE_TIMETICKS, TRANSLATE_OPAQUE, TRANSLATE_NOSUCHOBJECT, TRANSLATE_NOSUCHINSTANCE, TRANSLATE_ENDOFMIBVIEW, TRANSLATE_UNSIGNED, TRANSLATE_ALL

Tags

:asn1

INTEGER, INTEGER32, OCTET_STRING, NULL, OBJECT_IDENTIFIER, SEQUENCE, IPADDRESS, COUNTER, COUNTER32, GAUGE, GAUGE32, UNSIGNED32, TIMETICKS, OPAQUE, COUNTER64, NOSUCHOBJECT, NOSUCHINSTANCE, ENDOFMIBVIEW, GET_REQUEST, GET_NEXT_REQUEST, GET_RESPONSE, SET_REQUEST, TRAP, GET_BULK_REQUEST, INFORM_REQUEST, SNMPV2_TRAP

:debug

&snmp_debug, DEBUG_ALL, DEBUG_NONE, DEBUG_MESSAGE, DEBUG_TRANSPORT, DEBUG_DISPATCHER, DEBUG_PROCESSING, DEBUG_SECURITY

:generictrap

COLD_START, WARM_START, LINK_DOWN, LINK_UP, AUTHENTICATION_FAILURE, EGP_NEIGHBOR_LOSS, ENTERPRISE_SPECIFIC

:snmp

&snmp_debug, &snmp_dispatcher, &oid_base_match, &oid_lex_sort, &ticks_to_time, SNMP_VERSION_1, SNMP_VERSION_2C, SNMP_VERSION_3, SNMP_PORT, SNMP_TRAP_PORT

:translate

TRANSLATE_NONE, TRANSLATE_OCTET_STRING, TRANSLATE_NULL, TRANSLATE_TIMETICKS, TRANSLATE_OPAQUE, TRANSLATE_NOSUCHOBJECT, TRANSLATE_NOSUCHINSTANCE, TRANSLATE_ENDOFMIBVIEW, TRANSLATE_UNSIGNED, TRANSLATE_ALL

:ALL

All of the above exportable items.

1.7 EXAMPLES

1. Blocking SNMPv1 get-request for sysUpTime

This example gets the sysUpTime from a remote host.

```
#! /usr/local/bin/perl

use strict;

use Net::SNMP;

my ($session, $error) = Net::SNMP->session(
   -hostname  => shift || 'localhost',
   -community => shift || 'public',
   -port      => shift || 161
);
```

```
if (!defined($session)) {
   printf("ERROR: %s.\n", $error);
   exit 1;
}

my $sysUpTime = '1.3.6.1.2.1.1.3.0';

my $result = $session->get_request(
   -varbindlist => [$sysUpTime]
);

if (!defined($result)) {
   printf("ERROR: %s.\n", $session->error);
   $session->close;
   exit 1;
}

printf("sysUpTime for host '%s' is %s\n",
   $session->hostname, $result->{$sysUpTime}
);

$session->close;

exit 0;
```

2. Blocking SNMPv3 set-request of sysContact

This example sets the sysContact information on the remote host to "Help Desk x911". The named arguments passed to the `session()` constructor are for the demonstration of syntax only. These parameters will need to be set according to the SNMPv3 parameters of the remote host used by the script.

```
#! /usr/local/bin/perl

use strict;

use Net::SNMP;

my ($session, $error) = Net::SNMP->session(
   -hostname     => 'myv3host.company.com',
   -version      => 'snmpv3',
   -username     => 'myv3Username',
   -authkey      => '0x05c7fbde31916f64da4d5b77156bdfa7',
   -authprotocol => 'md5',
   -privkey      => '0x93725fd3a02a48ce02df4e065a1c1746'
);

if (!defined($session)) {
   printf("ERROR: %s.\n", $error);
   exit 1;
}

my $sysContact = '1.3.6.1.2.1.1.4.0';

my $result = $session->set_request(
   -varbindlist => [$sysContact, OCTET_STRING, 'Help Desk x911']
);

if (!defined($result)) {
   printf("ERROR: %s.\n", $session->error);
   $session->close;
   exit 1;
}
```

```
printf("sysContact for host '%s' set to '%s'\n",
   $session->hostname, $result->{$sysContact}
);

$session->close;

exit 0;
```

3. Non-blocking SNMPv2c get-bulk-request for ifTable

This example gets the contents of the ifTable by sending get-bulk-requests until the responses are no longer part of the ifTable. The ifTable can also be retrieved using the `get_table()` method.

```
#! /usr/local/bin/perl

use strict;

use Net::SNMP qw(:snmp);

my ($session, $error) = Net::SNMP->session(
   -version     => 'snmpv2c',
   -nonblocking => 1,
   -hostname    => shift || 'localhost',
   -community   => shift || 'public',
   -port        => shift || 161
);

if (!defined($session)) {
   printf("ERROR: %s.\n", $error);
   exit 1;
}

my $ifTable = '1.3.6.1.2.1.2.2';

my $result = $session->get_bulk_request(
   -callback       => [\&table_cb, {}],
   -maxrepetitions => 10,
   -varbindlist    => [$ifTable]
);

if (!defined($result)) {
   printf("ERROR: %s.\n", $session->error);
   $session->close;
   exit 1;
}

snmp_dispatcher();

$session->close;

exit 0;

sub table_cb
{
   my ($session, $table) = @_;

   if (!defined($session->var_bind_list)) {

      printf("ERROR: %s\n", $session->error);

   } else {
```

```
      # Loop through each of the OIDs in the response and assign
      # the key/value pairs to the anonymous hash that is passed
      # to the callback.  Make sure that we are still in the table
      # before assigning the key/values.

      my $next;

      foreach my $oid (oid_lex_sort(keys(%{$session->var_bind_list}))) {
         if (!oid_base_match($ifTable, $oid)) {
            $next = undef;
            last;
         }
         $next = $oid;
         $table->{$oid} = $session->var_bind_list->{$oid};
      }

      # If $next is defined we need to send another request
      # to get more of the table.

      if (defined($next)) {

         $result = $session->get_bulk_request(
            -callback       => [\&table_cb, $table],
            -maxrepetitions => 10,
            -varbindlist    => [$next]
         );

         if (!defined($result)) {
            printf("ERROR: %s\n", $session->error);
         }

      } else {

         # We are no longer in the table, so print the results.

         foreach my $oid (oid_lex_sort(keys(%{$table}))) {
            printf("%s => %s\n", $oid, $table->{$oid});
         }

      }
   }
}
```

4. Non-blocking SNMPv1 get-request for sysUpTime on multiple hosts

This example polls several hosts for their sysUpTime using non-blocking objects and reports a warning if this value is less than the value from the last poll.

```
#! /usr/local/bin/perl

use strict;

use Net::SNMP qw(snmp_dispatcher ticks_to_time);

# List of hosts to poll

my @HOSTS = qw(1.1.1.1 1.1.1.2 localhost);

# Poll interval (in seconds).  This value should be greater
# than the number of retries plus one, times the timeout value.

my $INTERVAL  = 60;

# Maximum number of polls, including the initial poll.
```

```
my $MAX_POLLS = 10;

my $sysUpTime = '1.3.6.1.2.1.1.3.0';

# Create a session for each host and queue the first get-request.

foreach my $host (@HOSTS) {

   my ($session, $error) = Net::SNMP->session(
      -hostname    => $host,
      -nonblocking => 0x1,   # Create non-blocking objects
      -translate   => [
         -timeticks => 0x0   # Turn off so sysUpTime is numeric
      ]
   );
   if (!defined($session)) {
      printf("ERROR: %s.\n", $error);
      exit 1;
   }

   # Queue the get-request, passing references to variables that
   # will be used to store the last sysUpTime and the number of
   # polls that this session has performed.

   my ($last_uptime, $num_polls) = (0, 0);

   $session->get_request(
       -varbindlist => [$sysUpTime],
       -callback    => [
          \&validate_sysUpTime_cb, \$last_uptime, \$num_polls
       ]
   );

}

# Define a reference point for all of the polls
my $EPOC = time();

# Enter the event loop
snmp_dispatcher();

exit 0;

sub validate_sysUpTime_cb
{
   my ($session, $last_uptime, $num_polls) = @_;

   if (!defined($session->var_bind_list)) {

      printf("%-15s  ERROR: %s\n", $session->hostname, $session->error);

   } else {

      # Validate the sysUpTime

      my $uptime = $session->var_bind_list->{$sysUpTime};

      if ($uptime < ${$last_uptime}) {
         printf("%-15s  WARNING: %s is less than %s\n",
            $session->hostname,
            ticks_to_time($uptime),
            ticks_to_time(${$last_uptime})
         );
      } else {
         printf("%-15s  Ok (%s)\n",
            $session->hostname, ticks_to_time($uptime)
         );
      }
```

```
      # Store the new sysUpTime
      ${$last_uptime} = $uptime;

   }

   # Queue the next message if we have not reached $MAX_POLLS.
   # Since we do not provide a -callback argument, the same
   # callback and it's original arguments will be used.

   if (++${$num_polls} < $MAX_POLLS) {
      my $delay = (($INTERVAL * ${$num_polls}) + $EPOC) - time();
      $session->get_request(
         -delay       => ($delay >= 0) ? $delay : 0,
         -varbindlist => [$sysUpTime]
      );
   }

   $session->error_status;
}
```

1.8 REQUIREMENTS

- The Net::SNMP module uses syntax that is not supported in versions of Perl earlier than v5.6.0.
- The non-core modules *Crypt::DES*, *Digest::MD5*, *Digest::SHA1*, and *Digest::HMAC* are required to support SNMPv3.
- In order to support the AES Cipher Algorithm as a SNMPv3 privacy protocol, the non-core module *Crypt::Rijndael* is needed.

1.9 AUTHOR

David M. Town <dtown@cpan.org>

1.10 ACKNOWLEDGMENTS

The original concept for this module was based on *SNMP_Session.pm* written by Simon Leinen <simon@switch.ch>.

The Abstract Syntax Notation One (ASN.1) encode and decode methods were derived by example from the CMU SNMP package whose copyright follows: Copyright (c) 1988, 1989, 1991, 1992 by Carnegie Mellon University. All rights reserved.

1.11 COPYRIGHT

Chapter 2

Net::LDAP

2.1 NAME

Net::LDAP - Lightweight Directory Access Protocol

2.2 SYNOPSIS

```
use Net::LDAP;

$ldap = Net::LDAP->new( 'ldap.bigfoot.com' ) or die "$@";

$mesg = $ldap->bind ;    # an anonymous bind

$mesg = $ldap->search( # perform a search
                       base   => "c=US",
                       filter => "(&(sn=Barr) (o=Texas Instruments))"
                     );

$mesg->code && die $mesg->error;

foreach $entry ($mesg->all_entries) { $entry->dump; }

$mesg = $ldap->unbind;   # take down session

$ldap = Net::LDAP->new( 'ldap.umich.edu' );

# bind to a directory with dn and password
$mesg = $ldap->bind( 'cn=root, o=University of Michigan, c=us',
                     password => 'secret'
                   );

$result = $ldap->add( 'cn=Barbara Jensen, o=University of Michigan, c=US',
                      attr => [
                        'cn'   => ['Barbara Jensen', 'Barbs Jensen'],
                        'sn'   => 'Jensen',
                        'mail' => 'b.jensen@umich.edu',
                        'objectclass' => ['top', 'person',
                                          'organizationalPerson',
                                          'inetOrgPerson' ],
                      ]
                    );

$result->code && warn "failed to add entry: ", $result->error ;
$mesg = $ldap->unbind;  # take down session
```

2.3 DESCRIPTION

Net::LDAP is a collection of modules that implements a LDAP services API for Perl programs. The module may be used to search directories or perform maintenance functions such as adding, deleting or modifying entries.

This document assumes that the reader has some knowledge of the LDAP protocol.

2.4 CONSTRUCTOR

new (HOST, OPTIONS)

Creates a new **Net::LDAP** object and opens a connection to the named host.

`HOST` may be a host name or an IP number. TCP port may be specified after the host name followed by a colon (such as localhost:10389). The default TCP port for LDAP is 389.

You can also specify a URI, such as 'ldaps://127.0.0.1:666' or 'ldapi://%2fvar%2flib%2fldap_sock'. Note that '%2f's in the LDAPI socket path will be translated into '/'. This is to support LDAP query options like base, search etc. although the query part of the URI will be ignored in this context. If port was not specified in the URI, the default is either 389 or 636 for 'LDAP' and 'LDAPS' schemes respectively.

`HOST` may also be a reference to an array of hosts, host-port pairs or URIs to try. Each will be tried in order until a connection is made. Only when all have failed will the result of `undef` be returned.

port => N

Port to connect to on the remote server. May be overridden by `HOST`.

timeout => N

Timeout passed to *IO::Socket* when connecting the remote server. (Default: 120)

multihomed => N

Will be passed to *IO::Socket* as the `MultiHomed` parameter when connecting to the remote server

debug => N

Set the debug level. See the `debug` method for details.

async => 1

Perform all operations asynchronously.

onerror => 'die' | 'warn' | undef | sub { ... }

In synchronous mode, change what happens when an error is detected.

'die'

Net::LDAP will croak whenever an error is detected.

'warn'

Net::LDAP will warn whenever an error is detected.

undef

Net::LDAP will warn whenever an error is detected and `-w` is in effect. The method that was called will return `undef`.

sub { ... }

The given sub will be called in a scalar context with a single argument, the result message. The value returned will be the return value for the method that was called.

version => N

Set the protocol version being used (default is LDAPv3). This is useful if you want to talk to an old server and therefore have to use LDAPv2.

Example

```
$ldap = Net::LDAP->new( 'remote.host', async => 1 );
```

LDAPS connections have some extra valid options, see the start_tls|start_tls method for details. Note the default value for 'sslversion' for LDAPS is 'sslv2/3', and the default port for LDAPS is 636.

For LDAPI connections, HOST is actually the location of a UNIX domain socket to connect to. The default location is '/var/lib/ldapi'.

2.5 METHODS

Each of the following methods take as arguments some number of fixed parameters followed by options, these options are passed in a named fashion, for example

```
$mesg = $ldap->bind( "cn=me,o=example", password => "mypasswd");
```

The return value from these methods is an object derived from the *Net::LDAP::Message* class. The methods of this class allow you to examine the status of the request.

abandon (ID, OPTIONS)

Abandon a previously issued request. `ID` may be a number or an object which is a sub-class of *Net::LDAP::Message*, returned from a previous method call.

control => CONTROL

control => [CONTROL, ...]

See CONTROLS below

callback => CALLBACK

See CALLBACKS below

Example

```
$res = $ldap->search( @search_args );

$mesg = $ldap->abandon( $res ); # This could be written as $res->abandon
```

add (DN, OPTIONS)

Add a new entry to the directory. `DN` can be either a *Net::LDAP::Entry* object or a string.

attrs => [ATTR => VALUE, ...]

`VALUE` should be a string if only a single value is wanted, or a reference to an array of strings if multiple values are wanted.

This argument is not used if `DN` is a *Net::LDAP::Entry* object.

control => CONTROL

control => [CONTROL, ...]

See CONTROLS below

callback => CALLBACK

See CALLBACKS below

Example

```
# $entry is an object of class Net::LDAP::Entry
$mesg = $ldap->add( $entry );

$mesg = $ldap->add( $dn,
                    attrs => [
                      name  => 'Graham Barr',
                      attr  => 'value1',
                      attr  => 'value2',
                      multi => [qw(value1 value2)]
                    ]
                  );
```

bind (DN, OPTIONS)

Bind (log in) to the server. `DN` is the DN to bind with. An anonymous bind may be done by calling bind without any arguments.

control => CONTROL

control => [CONTROL, ...]

See CONTROLS below

callback => CALLBACK

See CALLBACKS below

noauth | anonymous => 1

Bind without any password. The value passed with this option is ignored.

password => PASSWORD

Bind with the given password.

sasl => SASLOBJ

Bind using a SASL mechanism. The argument given should be a sub-class of *Authen::SASL*.

Example

```
$mesg = $ldap->bind; # Anonymous bind

$mesg = $ldap->bind( $dn, password => $password );

# $sasl is an object of class Authen::SASL
$mesg = $ldap->bind( $dn, sasl => $sasl, version => 3 );
```

compare (DN, OPTIONS)

Compare values in an attribute in the entry given by DN on the server. DN may be a string or a *Net::LDAP::Entry* object.

attr => ATTR

The name of the attribute to compare.

value => VALUE

The value to compare with.

control => CONTROL

control => [CONTROL, ...]

See CONTROLS below.

callback => CALLBACK

See CALLBACKS below.

Example

```
$mesg = $ldap->compare( $dn,
                        attr  => 'cn',
                        value => 'Graham Barr'
                      );
```

delete (DN, OPTIONS)

Delete the entry given by DN from the server. DN may be a string or a *Net::LDAP::Entry* object.

control => CONTROL

control => [CONTROL, ...]

See CONTROLS below.

callback => CALLBACK

See CALLBACKS below.

Example

```
$mesg = $ldap->delete( $dn );
```

moddn (DN, OPTIONS)

Rename the entry given by `DN` on the server. `DN` may be a string or a *Net::LDAP::Entry* object.

newrdn => RDN
This value should be a new RDN to assign to `DN`.

deleteoldrdn => 1
This option should be passwd if the existing RDN is to be deleted.

newsuperior => NEWDN
If given this value should be the DN of the new superior for `DN`.

control => CONTROL

control => [CONTROL, ...]
See CONTROLS below.

callback => CALLBACK
See CALLBACKS below.

Example

```
$mesg = $ldap->moddn( $dn, newrdn => 'cn=Graham Barr' );
```

modify (DN, OPTIONS)

Modify the contents of the entry given by `DN` on the server. `DN` may be a string or a *Net::LDAP::Entry* object.

add => { ATTR => VALUE, ... }
Add more attributes or values to the entry. `VALUE` should be a string if only a single value is wanted in the attribute, or a reference to an array of strings if multiple values are wanted.

delete => [ATTR, ...]
Delete complete attributes from the entry.

delete => { ATTR => VALUE, ... }
Delete individual values from an attribute. `VALUE` should be a string if only a single value is being deleted from the attribute, or a reference to an array of strings if multiple values are being deleted.

replace => { ATTR => VALUE, ... }
Replace any existing values in each given attribute with `VALUE`. `VALUE` should be a string if only a single value is wanted in the attribute, or a reference to an array of strings if multiple values are wanted. A reference to an empty array will remove the entire attribute.

changes => [OP => [ATTR => VALUE], ...]
This is an alternative to **add**, **delete** and **replace** where the whole operation can be given in a single argument. `OP` should be **add**, **delete** or **replace**. `VALUE` should be either a string or a reference to an array of strings, as before.
Use this form if you want to control the order in which the operations will be performed.

control => CONTROL

control => [CONTROL, ...]
See CONTROLS below.

callback => CALLBACK
See CALLBACKS below.

Example

```
$mesg = $ldap->modify( $dn, add => { sn => 'Barr' } );

$mesg = $ldap->modify( $dn, delete => [qw(faxNumber)] );

$mesg = $ldap->modify( $dn, delete => { 'telephoneNumber' => '911' } );
```

```
$mesg = $ldap->modify( $dn, replace => { 'mail' => 'gbarr@pobox.com' } );

$mesg = $ldap->modify( $dn,
                       changes => [
                           # add sn=Barr
                         add     => [ sn => 'Barr' ],
                           # delete all fax numbers
                         delete  => [ faxNumber => []],
                           # delete phone number 911
                         delete  => [ telephoneNumber => ['911']],
                           # change email address
                         replace => [ mail => 'gbarr@pobox.com']
                       ]
                     );
```

search (OPTIONS)

Search the directory using a given filter. This can be used to read attributes from a single entry, from entries immediately below a particular entry, or a whole subtree of entries.

The result is an object of class *Net::LDAP::Search*.

base => DN

The DN that is the base object entry relative to which the search is to be performed.

scope => 'base' | 'one' | 'sub'

By default the search is performed on the whole tree below the specified base object. This maybe changed by specifying a `scope` parameter with one of the following values:

base

Search only the base object.

one

Search the entries immediately below the base object.

sub

Search the whole tree below (and including) the base object. This is the default.

deref => 'never' | 'search' | 'find' | 'always'

By default aliases are dereferenced to locate the base object for the search, but not when searching subordinates of the base object. This may be changed by specifying a `deref` parameter with one of the following values:

never

Do not dereference aliases in searching or in locating the base object of the search.

search

Dereference aliases in subordinates of the base object in searching, but not in locating the base object of the search.

find

Dereference aliases in locating the base object of the search, but not when searching subordinates of the base object. This is the default.

always

Dereference aliases both in searching and in locating the base object of the search.

sizelimit => N

A sizelimit that restricts the maximum number of entries to be returned as a result of the search. A value of 0, and the default, means that no restriction is requested. Servers may enforce a maximum number of entries to return.

timelimit => N

A timelimit that restricts the maximum time (in seconds) allowed for a search. A value of 0 (the default), means that no timelimit will be requested.

typesonly => 1

Only attribute types (no values) should be returned. Normally attribute types and values are returned.

filter => FILTER

A filter that defines the conditions an entry in the directory must meet in order for it to be returned by the search. This may be a string or a *Net::LDAP::Filter* object. Values inside filters may need to be escaped to avoid security problems; see *Net::LDAP::Filter* for a definition of the filter format, including the escaping rules.

attrs => [ATTR, ...]

A list of attributes to be returned for each entry that matches the search filter.

If not specified, then the server will return the attributes that are specified as accessible by default given your bind credentials.

Certain additional attributes such as "createTimestamp" and other operational attributes may also be available for the asking:

```
$mesg = $ldap->search( ... ,
                       attrs => ['createTimestamp']
                     );
```

To retrieve the default attributes and additional ones, use '*'.

```
$mesg = $ldap->search( ... ,
                       attrs => ['*', 'createTimestamp']
                     );
```

To retrieve no attributes (the server only returns the DNs of matching entries), use '1.1':

```
$mesg = $ldap->search( ... ,
                       attrs => ['1.1']
                     );
```

control => CONTROL

control => [CONTROL, ...]

See CONTROLS below.

callback => CALLBACK

See CALLBACKS below.

Example

```
$mesg = $ldap->search(
                        base   => $base_dn,
                        scope  => 'sub',
                        filter => '(|(objectclass=rfc822mailgroup)(sn=jones))'
                      );

Net::LDAP::LDIF->new( \*STDOUT,"w" )->write( $mesg->entries );
```

start_tls (OPTIONS)

Calling this method will convert the existing connection to using Transport Layer Security (TLS), which provides an encrypted connection. This is *only* possible if the connection uses LDAPv3, and requires that the server advertizes support for LDAP_EXTENSION_START_TLS. Use supported_extension in *Net::LDAP::RootDSE* to check this.

verify => 'none' | 'optional' | 'require'

How to verify the server's certificate:

none

The server may provide a certificate but it will not be checked - this may mean you are be connected to the wrong server

optional

Verify only when the server offers a certificate

require

The server must provide a certificate, and it must be valid.

If you set verify to optional or require, you must also set either cafile or capath. The most secure option is **require**.

sslversion => 'sslv2' | **'sslv3'** | **'sslv2/3'** | **'tlsv1'**

This defines the version of the SSL/TLS protocol to use. Defaults to **'tlsv1'**.

ciphers => CIPHERS

Specify which subset of cipher suites are permissible for this connection, using the standard OpenSSL string format. The default value is **'ALL'**, which permits all ciphers, even those that don't encrypt.

clientcert => '/path/to/cert.pem'

clientkey => '/path/to/key.pem'

keydecrypt => sub { ... }

If you want to use the client to offer a certificate to the server for SSL authentication (which is not the same as for the LDAP Bind operation) then set clientcert to the user's certificate file, and clientkey to the user's private key file. These files must be in PEM format.

If the private key is encrypted (highly recommended) then keydecrypt should be a subroutine that returns the decrypting key. For example:

```
$ldap = Net::LDAP->new( 'myhost.example.com', version => 3 );
$mesg = $ldap->start_tls(
                           verify => 'require',
                           clientcert => 'mycert.pem',
                           clientkey => 'mykey.pem',
                           keydecrypt => sub { 'secret'; },
                           capath => '/usr/local/cacerts/'
                         );
```

capath => '/path/to/servercerts/'

cafile => '/path/to/servercert.pem'

When verifying the server's certificate, either set capath to the pathname of the directory containing CA certificates, or set cafile to the filename containing the certificate of the CA who signed the server's certificate. These certificates must all be in PEM format.

The directory in 'capath' must contain certificates named using the hash value of the certificates' subject names. To generate these names, use OpenSSL like this in Unix:

```
ln -s cacert.pem `openssl x509 -hash -noout < cacert.pem`.0
```

(assuming that the certificate of the CA is in cacert.pem.)

unbind ()

The unbind method does not take any parameters and will unbind you from the server. Some servers may allow you to re-bind or perform other operations after unbinding. If you wish to switch to another set of credentials while continuing to use the same connection, re-binding with another DN and password, without unbind-ing, will generally work.

Example

```
$mesg = $ldap->unbind;
```

The following methods are for convenience, and do not return `Net::LDAP::Message` objects.

async (VALUE)

If `VALUE` is given the async mode will be set. The previous value will be returned. The value is *true* if LDAP operations are being performed asynchronously.

certificate ()

Returns an X509_Certificate object containing the server's certificate. See the IO::Socket::SSL documentation for information about this class.

For example, to get the subject name (in a peculiar OpenSSL-specific format, different from RFC 1779 and RFC 2253) from the server's certificate, do this:

```
print "Subject DN: " . $ldaps->certificate->subject_name . "\n";
```

cipher ()

Returns the cipher mode being used by the connection, in the string format used by OpenSSL.

debug (VALUE)

If `VALUE` is given the debug bit-value will be set. The previous value will be returned. Debug output will be sent to `STDERR`. The bits of this value are:

```
1   Show outgoing packets (using asn_hexdump).
2   Show incoming packets (using asn_hexdump).
4   Show outgoing packets (using asn_dump).
8   Show incoming packets (using asn_dump).
```

The default value is 0.

disconnect ()

Disconnect from the server

root_dse (OPTIONS)

The root_dse method retrieves cached information from the server's rootDSE.

attrs => [ATTR, ...]

A reference to a list of attributes to be returned. If not specified, then the following attributes will be requested

```
subschemaSubentry
namingContexts
altServer
supportedExtension
supportedControl
supportedSASLMechanisms
supportedLDAPVersion
```

The result is an object of class *Net::LDAP::RootDSE*.

Example

```
my $root = $ldap->root_dse;
# get naming Context
$root->get_value( 'namingContext', asref => 1 );
# get supported LDAP versions
$root->supported_version;
```

As the root DSE may change in certain circumstances - for instance when you change the connection using start_tls - you should always use the root_dse method to return the most up-to-date copy of the root DSE.

schema (OPTIONS)

Read schema information from the server.

The result is an object of class *Net::LDAP::Schema*. Read this documentation for further information about methods that can be performed with this object.

dn => DN

If a DN is supplied, it will become the base object entry from which the search for schema information will be conducted. If no DN is supplied the base object entry will be determined from the rootDSE entry.

Example

```
my $schema = $ldap->schema;
# get objectClasses
@ocs = $schema->all_objectclasses;
# Get the attributes
@atts = $schema->all_attributes;
```

socket ()

Returns the underlying `IO::Socket` object being used.

sync (MESG)

Wait for a given `MESG` request to be completed by the server. If no `MESG` is given, then wait for all outstanding requests to be completed.

Returns an error code defined in *Net::LDAP::Constant*.

version ()

Returns the version of the LDAP protocol that is being used.

2.6 CONTROLS

Many of the methods described above accept a control option. This allows the user to pass controls to the server as described in LDAPv3.

A control is a reference to a HASH and should contain the three elements below. If any of the controls are blessed then the method `to_asn` will be called which should return a reference to a HASH containing the three elements described below.

type => OID

This element must be present and is the name of the type of control being requested.

critical => FLAG

critical is optional and should be a boolean value, if it is not specified then it is assumed to be *false*.

value => VALUE

If the control being requested requires a value then this element should hold the value for the server.

2.7 CALLBACKS

Most of the above commands accept a callback option. This option should be a reference to a subroutine. This subroutine will be called for each packet received from the server as a response to the request sent.

When the subroutine is called the first argument will be the *Net::LDAP::Message* object which was returned from the method.

If the request is a search then multiple packets can be received from the server. Each entry is received as a separate packet. For each of these the subroutine will be called with a *Net::LDAP::Entry* object as the second argument.

During a search the server may also send a list of references. When such a list is received then the subroutine will be called with a *Net::LDAP::Reference* object as the second argument.

2.8 LDAP ERROR CODES

Net::LDAP also exports constants for the error codes that can be received from the server, see *Net::LDAP::Constant*.

2.9 SEE ALSO

Net::LDAP::Constant, *Net::LDAP::Control*, *Net::LDAP::Entry*, *Net::LDAP::Filter*, *Net::LDAP::Message*, *Net::LDAP::Reference*, *Net::LDAP::Search*, *Net::LDAP::RFC*

The homepage for the perl-ldap modules can be found at http://perl-ldap.sourceforge.net/

2.10 ACKNOWLEDGEMENTS

This document is based on a document originally written by Russell Fulton <r.fulton@auckland.ac.nz>.

Chris Ridd <chris.ridd@isode.com> for the many hours spent testing and contribution of the ldap* command line utilities.

2.11 MAILING LIST

A discussion mailing list is hosted by sourceforge at <perl-ldap@perl.org> No subscription is necessary!

2.12 BUGS

We hope you do not find any, but if you do please report them to the mailing list.

If you have a patch, please send it as an attachment to the mailing list.

2.13 AUTHOR

Graham Barr <gbarr@pobox.com>

2.14 COPYRIGHT

$Id: LDAP.pod,v 1.35 2003/11/10 18:31:04 chrisridd Exp $

Chapter 3

Net::LDAP::Entry

3.1 NAME

Net::LDAP::Entry - An LDAP entry object

3.2 SYNOPSIS

```
use Net::LDAP;

$ldap = Net::LDAP->new ( $host );
$mesg = $ldap->search ( @search_args );

my $max = $mesg->count;
for ( $i = 0 ; $i < $max ; $i++ ) {
  my $entry = $mesg->entry ( $i );
  foreach my $attr ( $entry->attributes ) {
    print join( "\n ", $attr, $entry->get_value( $attr ) ), "\n";
  }
}

# or

use Net::LDAP::Entry;

$entry = Net::LDAP::Entry->new;

$entry->add (
  attr1 => 'value1',
  attr2 => [ qw(value1 value2) ]
);

$entry->delete ( 'unwanted' );

$entry->replace (
  attr1 => 'newvalue'
  attr2 => [ qw(new values) ]
);

$entry->update ( $ldap ); # update directory server

$entry2 = $entry->clone; # copies entry
```

3.3 DESCRIPTION

The **Net::LDAP::Entry** object represents a single entry in the directory. It is a container for attribute-value pairs.

A **Net::LDAP::Entry** object can be used in two situations. The first and probably most common use is in the result of a search to the directory server.

The other is where a new object is created locally and then a single command is sent to the directory server to add, modify or replace an entry. Entries for this purpose can also be created by reading an LDIF file with the *Net::LDAP::LDIF* module.

3.4 CONSTRUCTORS

new ()

Create a new entry object with the changetype set to `'add'`

clone ()

Returns a copy of the **Net::LDAP::Entry** object.

3.5 METHODS

add (ATTR => VALUE, ...)

Add more attributes or values to the entry. Each `VALUE` should be a string if only a single value is wanted in the attribute, or a reference to an array of strings if multiple values are wanted. The values given will be added to the values which already exist for the given attributes.

```
$entry->add ( 'sn' => 'Barr' );

$entry->add ( 'street' => [ '1 some road','nowhere' ] );
```

NOTE: these changes are local to the client and will not appear on the directory server until the `update` method is called.

attributes (OPTIONS)

Return a list of attributes in this entry

nooptions => 1

Return a list of the attribute names excluding any options. For example for the entry

```
name: Graham Barr
name;en-us: Bob
jpeg;binary: **binary data**
```

then

```
@values = $entry->attributes;
print "default: @values\n";

@values = $entry->attributes ( nooptions => 1 );
print "nooptions: @values\n";
```

will output

```
default: name name;en-us jpeg;binary
nooptions: name jpeg
```

changetype ()

Returns the type of operation that would be performed when the update method is called.

changetype (TYPE)

Set the type of operation that will be performed when the update method is called to `TYPE`.

Possible values for `TYPE` are

add

The update method will call the add method on the client object, which will result in the entry being added to the directory server.

delete

The update method will call the delete method on the client object, which will result in the entry being removed from the directory server.

modify

The update method will call the modify method on the client object, which will result in any changes that have been made locally being made to the entry on the directory server.

moddn/modrdn

The update method will call the moddn method on the client object, which will result in any DN changes that have been made locally being made to the entry on the directory server. These DN changes are specified by setting the entry attributes newrdn, deleteoldrdn, and (optionally) newsuperior.

delete ()

Delete the entry from the server on the next call to `update`.

delete (ATTR => [VALUE, ... , ...)]

Delete the values of given attributes from the entry. Values are references to arrays; passing a reference to an empty array is the same as passing `undef`, and will result in the entire attribute being deleted. For example:

```
$entry->delete ( 'mail' => [ 'foo.bar@example.com' ] );
$entry->delete ( 'description' => [ ], 'streetAddress' => [ ] );
```

NOTE: these changes are local to the client and will not appear on the directory server until the `update` method is called.

dn ()

Get the DN of the entry.

dn (DN)

Set the DN for the entry, and return the previous value.

NOTE: these changes are local to the client and will not appear on the directory server until the `update` method is called.

exists (ATTR)

Returns `TRUE` if the entry has an attribute called `ATTR`.

get_value (ATTR, OPTIONS)

Get the values for the attribute `ATTR`. In a list context returns all values for the given attribute, or the empty list if the attribute does not exist. In a scalar context returns the first value for the attribute or undef if the attribute does not exist.

alloptions => 1

The result will be a hash reference. The keys of the hash will be the options and the hash value will be the values for those attributes. For example if an entry had:

```
name: Graham Barr
name;en-us: Bob
```

Then a get for attribute "name" with alloptions set to a true value

```
$ref = $entry->get_value ( 'name', alloptions => 1 );
```

will return a hash reference that would be like

```
{
  ''       => [ 'Graham Barr' ],
  ';en-us' => [ 'Bob' ]
}
```

asref => 1

The result will be a reference to an array containing all the values for the attribute, or `undef` if the attribute does not exist.

```
$scalar = $entry->get_value ( 'name' );
```

$scalar will be the first value for the `name` attribute, or `undef` if the entry does not contain a `name` attribute.

```
$ref = $entry->get_value ( 'name', asref => 1 );
```

$ref will be a reference to an array, which will have all the values for the `name` attribute. If the entry does not have an attribute called `name` then $ref will be `undef`.

NOTE: In the interest of performance the array references returned by `get_value` are references to structures held inside the entry object. These values and thier contents should **NOT** be modified directly.

replace (ATTR => VALUE, ...)

Similar to `add`, except that the values given will replace any values that already exist for the given attributes.

NOTE: these changes are local to the client and will not appear on the directory server until the `update` method is called.

update (CLIENT)

Update the directory server with any changes that have been made locally to the attributes of this entry. This means any calls that have been made to add, replace or delete since the last call to changetype or update was made.

This method can also be used to modify the DN of the entry on the server, by specifying moddn or modrdn as the changetype, and setting the entry attributes newrdn, deleteoldrdn, and (optionally) newsuperior.

`CLIENT` is a `Net::LDAP` object where the update will be sent to.

The result will be an object of type *Net::LDAP::Message* as returned by the add, modify or delete method called on CLIENT.

3.6 SEE ALSO

Net::LDAP, *Net::LDAP::LDIF*

3.7 AUTHOR

Graham Barr <gbarr@pobox.com>.
Please report any bugs, or post any suggestions, to the perl-ldap mailing list <perl-ldap@perl.org>.

3.8 COPYRIGHT

$Id: Entry.pod,v 1.12 2003/08/01 19:00:39 chrisridd Exp $

Chapter 4

Net::LDAP::Message

4.1 NAME

Net::LDAP::Message - Message response from LDAP server

4.2 SYNOPSIS

```
use Net::LDAP;
```

4.3 DESCRIPTION

Net::LDAP::Message is a base class for the objects returned by the *Net::LDAP* methods abandon, add, bind, compare, delete, modify, moddn, search and unbind.

The sub-class *Net::LDAP::Search* returned by search also defines many methods.

If the *Net::LDAP* object is in async mode then all these methods, except `done`, will cause a wait until the request is completed.

4.4 METHODS

code ()

The code value in the result message from the server. Normally for a success zero will be returned. Constants for the result codes can be imported from the *Net::LDAP* or *Net::LDAP::Constant* module.

control ()

Return a list of controls that were returned from the server.

control (OID, ...)

Return a list of controls with the given OIDs that were returned from the server.

dn ()

The DN in the result message from the server.

done ()

Returns *true* if the request has been completed.

error ()

Returns the error message in the result message from the server. If the server did not include an error message, then the result of ldap_error_desc with the error code from the result message.

error_name ()

Returns the name of the error code in the result message from the server. See ldap_error_name|Net::LDAP::Util::ldap_error_name for a detailed description of the return value.

error_text ()

Returns the short text description of the error code in the result message from the server. See ldap_error_text|Net::LDAP::Util::ldap_error_text for a detailed description of the return value.

error_desc ()

Returns a long text description of the error code in the result message from the server. See ldap_error_desc|Net::LDAP::Util::ldap_error_desc for a detailed description of the return value.

is_error ()

Returns *true* if the result code is considered to be an error for the operation.

mesg_id ()

The message id of the request message sent to the server.

referrals ()

Returns a list of referrals from the result message.

server_error ()

The error message returned by the server, or `undef` if the server did not provide a message.

sync ()

Wait for the server to complete the request.

4.5 SEE ALSO

Net::LDAP, *Net::LDAP::Search*, *Net::LDAP::Constant*, *Net::LDAP::Util*

4.6 ACKNOWLEDGEMENTS

This document is based on a document originally written by Russell Fulton <r.fulton@auckland.ac.nz>.

4.7 AUTHOR

Graham Barr <gbarr@pobox.com>
Please report any bugs, or post any suggestions, to the perl-ldap mailing list <perl-ldap@perl.org>.

4.8 COPYRIGHT

$Id: Message.pod,v 1.8 2003/08/01 18:42:47 chrisridd Exp $

Chapter 5

Net::LDAP::Search

5.1 NAME

Net::LDAP::Search - Object returned by Net::LDAP search method

5.2 SYNOPSIS

```
use Net::LDAP;

$mesg = $ldap->search( @search_args );

@entries = $mesg->entries;
```

5.3 DESCRIPTION

A **Net::LDAP::Search** object is returned from the search method of a *Net::LDAP* object. It is a container object which holds the results of the search.

5.4 METHODS

Net::LDAP::Search inherits from *Net::LDAP::Message*, and so supports all methods defined in *Net::LDAP::Message*.

as_struct ()

Returns a reference to a HASH, where the keys are the DNs of the results and the values are HASH references. These second level HASHes hold the attributes such that the keys are the attribute names, in lowercase, and the values are references to an ARRAY holding the values.

This method will block until the whole search request has finished.

count ()

Returns the number of entries returned by the server.

This method will block until the whole search request has finished.

entry (INDEX)

Return the N'th entry, which will be a *Net::LDAP::Entry* object. If INDEX is greater than the total number of entries returned then `undef` will be returned.

This method will block until the search request has returned enough entries.

entries ()

Return an array of *Net::LDAP::Entry* objects that were returned from the server.

This method will block until the whole search request has finished.

pop_entry ()

Pop an entry from the internal list of *Net::LDAP::Entry* objects for this search. If there are no more entries then `undef` is returned.

This call will block if the list is empty, until the server returns another entry.

references ()

Return a list of references that the server returned. This will be a list of *Net::LDAP::Reference* objects.

sorted ()

Return a list Net::LDAP:Entry objects, sorted by their DNs.

sorted (ATTR, ...)

Return a list of *Net::LDAP::Entry* objects, sorted by the specified attributes. The attributes are compared in the order specified, each only being compared if all the prior attributes compare equal.

shift_entry ()

Shift an entry from the internal list of *Net::LDAP::Entry* objects for this search. If there are no more entries then `undef` is returned.

This call will block if the list is empty, until the server returns another entry.

5.5 SEE ALSO

Net::LDAP, *Net::LDAP::Message*, *Net::LDAP::Entry*, *Net::LDAP::Reference*

5.6 ACKNOWLEDGEMENTS

This document is based on a document originally written by Russell Fulton <r.fulton@auckland.ac.nz>.

5.7 AUTHOR

Graham Barr <gbarr@pobox.com>
Please report any bugs, or post any suggestions, to the perl-ldap mailing list <perl-ldap@perl.org>.

5.8 COPYRIGHT

$Id: Search.pod,v 1.7 2003/08/02 18:50:18 chrisridd Exp $

Chapter 6

Net::LDAP::Constant

6.1 NAME

Net::LDAP::Constant - Constants for use with Net::LDAP

6.2 SYNOPSIS

```
use Net::LDAP qw(LDAP_SUCCESS LDAP_PROTOCOL_ERROR);
```

6.3 DESCRIPTION

Net::LDAP::Constant exports constant subroutines for the following LDAP error codes.

Protocol Constants

LDAP_SUCCESS (0)

Operation completed without error

LDAP_OPERATIONS_ERROR (1)

Server encountered an internal error

LDAP_PROTOCOL_ERROR (2)

Unrecognized version number or incorrect PDU structure

LDAP_TIMELIMIT_EXCEEDED (3)

The time limit on a search operation has been exceeded

LDAP_SIZELIMIT_EXCEEDED (4)

The maximum number of search results to return has been exceeded.

LDAP_COMPARE_FALSE (5)

This code is returned when a compare request completes and the attribute value given is not in the entry specified

LDAP_COMPARE_TRUE (6)

This code is returned when a compare request completes and the attribute value given is in the entry specified

LDAP_AUTH_METHOD_NOT_SUPPORTED (7)

Unrecognized SASL mechanism name

LDAP_STRONG_AUTH_NOT_SUPPORTED (7)

Unrecognized SASL mechanism name

LDAP_STRONG_AUTH_REQUIRED (8)

The server requires authentication be performed with a SASL mechanism

LDAP_PARTIAL_RESULTS (9)

Returned to version 2 clients when a referral is returned. The response will contain a list of URL's for other servers.

LDAP_REFERRAL (10)

The server is referring the client to another server. The response will contain a list of URL's

LDAP_ADMIN_LIMIT_EXCEEDED (11)

The server has exceed the maximum number of entries to search while gathering a list of search result candidates

LDAP_UNAVAILABLE_CRITICAL_EXT (12)

A control or matching rule specified in the request is not supported by the server

LDAP_CONFIDENTIALITY_REQUIRED (13)

This result code is returned when confidentiality is required to perform a given operation

LDAP_SASL_BIND_IN_PROGRESS (14)

The server requires the client to send a new bind request, with the same SASL mechanism, to continue the authentication process

LDAP_NO_SUCH_ATTRIBUTE (16)

The request referenced an attribute that does not exist

LDAP_UNDEFINED_TYPE (17)

The request contains an undefined attribute type

LDAP_INAPPROPRIATE_MATCHING (18)

An extensible matching rule in the given filter does not apply to the specified attribute

LDAP_CONSTRAINT_VIOLATION (19)

The request contains a value which does not meet with certain constraints. This result can be returned as a consequence of

- The request was to add or modify a user password, and the password fails to meet the criteria the server is configured to check. This could be that the password is too short, or a recognizable word (e.g. it matches one of the attributes in the users entry) or it matches a previous password used by the same user.
- The request is a bind request to a user account that has been locked

LDAP_TYPE_OR_VALUE_EXISTS (20)

The request attempted to add an attribute type or value that already exists

LDAP_INVALID_SYNTAX (21)

Some part of the request contained an invalid syntax. It could be a search with an invalid filter or a request to modify the schema and the given schema has a bad syntax.

LDAP_NO_SUCH_OBJECT (32)

The server cannot find an object specified in the request

LDAP_ALIAS_PROBLEM (33)

Server encountered a problem while attempting to dereference an alias

LDAP_INVALID_DN_SYNTAX (34)

The request contained an invalid DN

LDAP_IS_LEAF (35)

The specified entry is a leaf entry

LDAP_ALIAS_DEREF_PROBLEM (36)

Server encountered a problem while attempting to dereference an alias

LDAP_INAPPROPRIATE_AUTH (48)

The server requires the client which had attempted to bind anonymously or without supplying credentials to provide some form of credentials

LDAP_INVALID_CREDENTIALS (49)

The wrong password was supplied or the SASL credentials could not be processed

LDAP_INSUFFICIENT_ACCESS (50)

The client does not have sufficient access to perform the requested operation

LDAP_BUSY (51)

The server is too busy to perform requested operation

LDAP_UNAVAILABLE (52)

The server in unavailable to perform the request, or the server is shutting down

LDAP_UNWILLING_TO_PERFORM (53)

The server is unwilling to perform the requested operation

LDAP_LOOP_DETECT (54)

The server was unable to perform the request due to an internal loop detected

LDAP_SORT_CONTROL_MISSING (60)

The search contained a "virtual list view" control, but not a server-side sorting control, which is required when a "virtual list view" is given.

LDAP_INDEX_RANGE_ERROR (61)

The search contained a control for a "virtual list view" and the results exceeded the range specified by the requested offsets.

LDAP_NAMING_VIOLATION (64)

The request violates the structure of the DIT

LDAP_OBJECT_CLASS_VIOLATION (65)

The request specifies a change to an existing entry or the addition of a new entry that does not comply with the servers schema

LDAP_NOT_ALLOWED_ON_NONLEAF (66)

The requested operation is not allowed on an entry that has child entries

LDAP_NOT_ALLOWED_ON_RDN (67)

The requested operation ill affect the RDN of the entry

LDAP_ALREADY_EXISTS (68)

The client attempted to add an entry that already exists. This can occur as a result of

- An add request was submitted with a DN that already exists
- A modify DN requested was submitted, where the requested new DN already exists
- The request is adding an attribute to the schema and an attribute with the given OID or name already exists

LDAP_NO_OBJECT_CLASS_MODS (69)

Request attempt to modify the object class of an entry that should not be modified

LDAP_RESULTS_TOO_LARGE (70)

The results of the request are to large

LDAP_AFFECTS_MULTIPLE_DSAS (71)

The requested operation needs to be performed on multiple servers where the requested operation is not permitted

LDAP_OTHER (80)

An unknown error has occurred

LDAP_SERVER_DOWN (81)

`Net::LDAP` cannot establish a connection or the connection has been lost

LDAP_LOCAL_ERROR (82)

An error occurred in `Net::LDAP`

LDAP_ENCODING_ERROR (83)

`Net::LDAP` encountered an error while encoding the request packet that would have been sent to the server

LDAP_DECODING_ERROR (84)

`Net::LDAP` encountered an error while decoding a response packet from the server.

LDAP_TIMEOUT (85)

`Net::LDAP` timeout while waiting for a response from the server

LDAP_AUTH_UNKNOWN (86)

The method of authentication requested in a bind request is unknown to the server

LDAP_FILTER_ERROR (87)

An error occurred while encoding the given search filter.

LDAP_USER_CANCELED (88)

The user canceled the operation

LDAP_PARAM_ERROR (89)

An invalid parameter was specified

LDAP_NO_MEMORY (90)

Out of memory error

LDAP_CONNECT_ERROR (91)

A connection to the server could not be established

LDAP_NOT_SUPPORTED (92)

An attempt has been made to use a feature not supported by Net::LDAP

LDAP_CONTROL_NOT_FOUND (93)

The controls required to perform the requested operation were not found.

LDAP_NO_RESULTS_RETURNED (94)

No results were returned from the server.

LDAP_MORE_RESULTS_TO_RETURN (95)

There are more results in the chain of results.

LDAP_CLIENT_LOOP (96)

A loop has been detected. For example when following referals.

LDAP_REFERRAL_LIMIT_EXCEEDED (97)

The referral hop limit has been exceeded.

Control OIDs

LDAP_CONTROL_SORTREQUEST (1.2.840.113556.1.4.473)
LDAP_CONTROL_SORTRESULT (1.2.840.113556.1.4.474)
LDAP_CONTROL_VLVREQUEST (2.16.840.1.113730.3.4.9)
LDAP_CONTROL_VLVRESPONSE (2.16.840.1.113730.3.4.10)
LDAP_CONTROL_PROXYAUTHENTICATION (2.16.840.1.113730.3.4.12)
LDAP_CONTROL_PAGED (1.2.840.113556.1.4.319)
LDAP_CONTROL_TREE_DELETE (1.2.840.113556.1.4.805)
LDAP_CONTROL_MATCHEDVALS (1.2.826.0.1.3344810.2.2)
LDAP_CONTROL_MANAGEDSAIT (2.16.840.1.113730.3.4.2)
LDAP_CONTROL_PERSISTENTSEARCH (2.16.840.1.113730.3.4.3)
LDAP_CONTROL_ENTRYCHANGE (2.16.840.1.113730.3.4.7)
LDAP_CONTROL_PWEXPIRED (2.16.840.1.113730.3.4.4)
LDAP_CONTROL_PWEXPIRING (2.16.840.1.113730.3.4.5)
LDAP_CONTROL_REFERRALS (1.2.840.113556.1.4.616)

Extension OIDs

Net::LDAP::Constant exports constant subroutines for the following LDAP extension OIDs.

LDAP_EXTENSION_START_TLS (1.3.6.1.4.1.1466.20037)

Indicates if the server supports the Start TLS extension (RFC-2830)

6.4 SEE ALSO

Net::LDAP, *Net::LDAP::Message*

6.5 AUTHOR

Graham Barr <gbarr@pobox.com>
Please report any bugs, or post any suggestions, to the perl-ldap mailing list <perl-ldap@perl.org>

6.6 COPYRIGHT

$Id: Constant.pm,v 1.9 2003/06/02 15:13:10 gbarr Exp $

Chapter 7

Perl Data Stuctures Cookbook

7.1 NAME

perldsc - Perl Data Structures Cookbook

7.2 DESCRIPTION

The single feature most sorely lacking in the Perl programming language prior to its 5.0 release was complex data structures. Even without direct language support, some valiant programmers did manage to emulate them, but it was hard work and not for the faint of heart. You could occasionally get away with the `$m{$AoA,$b}` notation borrowed from **awk** in which the keys are actually more like a single concatenated string `"$AoA$b"`, but traversal and sorting were difficult. More desperate programmers even hacked Perl's internal symbol table directly, a strategy that proved hard to develop and maintain–to put it mildly.

The 5.0 release of Perl let us have complex data structures. You may now write something like this and all of a sudden, you'd have an array with three dimensions!

```
for $x (1 .. 10) {
    for $y (1 .. 10) {
        for $z (1 .. 10) {
            $AoA[$x][$y][$z] =
                $x ** $y + $z;
        }
    }
}
```

Alas, however simple this may appear, underneath it's a much more elaborate construct than meets the eye!

How do you print it out? Why can't you say just `print @AoA`? How do you sort it? How can you pass it to a function or get one of these back from a function? Is it an object? Can you save it to disk to read back later? How do you access whole rows or columns of that matrix? Do all the values have to be numeric?

As you see, it's quite easy to become confused. While some small portion of the blame for this can be attributed to the reference-based implementation, it's really more due to a lack of existing documentation with examples designed for the beginner.

This document is meant to be a detailed but understandable treatment of the many different sorts of data structures you might want to develop. It should also serve as a cookbook of examples. That way, when you need to create one of these complex data structures, you can just pinch, pilfer, or purloin a drop-in example from here.

Let's look at each of these possible constructs in detail. There are separate sections on each of the following:

- arrays of arrays
- hashes of arrays
- arrays of hashes
- hashes of hashes

- more elaborate constructs

But for now, let's look at general issues common to all these types of data structures.

7.3 REFERENCES

The most important thing to understand about all data structures in Perl – including multidimensional arrays– is that even though they might appear otherwise, Perl `@ARRAY`s and `%HASH`es are all internally one-dimensional. They can hold only scalar values (meaning a string, number, or a reference). They cannot directly contain other arrays or hashes, but instead contain *references* to other arrays or hashes.

You can't use a reference to an array or hash in quite the same way that you would a real array or hash. For C or C++ programmers unused to distinguishing between arrays and pointers to the same, this can be confusing. If so, just think of it as the difference between a structure and a pointer to a structure.

You can (and should) read more about references in the perlref(1) man page. Briefly, references are rather like pointers that know what they point to. (Objects are also a kind of reference, but we won't be needing them right away–if ever.) This means that when you have something which looks to you like an access to a two-or-more-dimensional array and/or hash, what's really going on is that the base type is merely a one-dimensional entity that contains references to the next level. It's just that you can *use* it as though it were a two-dimensional one. This is actually the way almost all C multidimensional arrays work as well.

```
$array[7][12]                       # array of arrays
$array[7]{string}                   # array of hashes
$hash{string}[7]                    # hash of arrays
$hash{string}{'another string'}     # hash of hashes
```

Now, because the top level contains only references, if you try to print out your array in with a simple print() function, you'll get something that doesn't look very nice, like this:

```
@AoA = ( [2, 3], [4, 5, 7], [0] );
print $AoA[1][2];
7
print @AoA;
ARRAY(0x83c38)ARRAY(0x8b194)ARRAY(0x8b1d0)
```

That's because Perl doesn't (ever) implicitly dereference your variables. If you want to get at the thing a reference is referring to, then you have to do this yourself using either prefix typing indicators, like `${$blah}`, `@{$blah}`, `@{$blah[$i]}`, or else postfix pointer arrows, like `$a->[3]`, `$h->{fred}`, or even `$ob->method()->[3]`.

7.4 COMMON MISTAKES

The two most common mistakes made in constructing something like an array of arrays is either accidentally counting the number of elements or else taking a reference to the same memory location repeatedly. Here's the case where you just get the count instead of a nested array:

```
for $i (1..10) {
    @array = somefunc($i);
    $AoA[$i] = @array;      # WRONG!
}
```

That's just the simple case of assigning an array to a scalar and getting its element count. If that's what you really and truly want, then you might do well to consider being a tad more explicit about it, like this:

```
for $i (1..10) {
    @array = somefunc($i);
    $counts[$i] = scalar @array;
}
```

Here's the case of taking a reference to the same memory location again and again:

```
for $i (1..10) {
    @array = somefunc($i);
    $AoA[$i] = \@array;     # WRONG!
}
```

So, what's the big problem with that? It looks right, doesn't it? After all, I just told you that you need an array of references, so by golly, you've made me one!

Unfortunately, while this is true, it's still broken. All the references in @AoA refer to the *very same place*, and they will therefore all hold whatever was last in @array! It's similar to the problem demonstrated in the following C program:

```
#include <pwd.h>
main() {
    struct passwd *getpwnam(), *rp, *dp;
    rp = getpwnam("root");
    dp = getpwnam("daemon");

    printf("daemon name is %s\nroot name is %s\n",
            dp->pw_name, rp->pw_name);
}
```

Which will print

```
daemon name is daemon
root name is daemon
```

The problem is that both `rp` and `dp` are pointers to the same location in memory! In C, you'd have to remember to malloc() yourself some new memory. In Perl, you'll want to use the array constructor `[]` or the hash constructor {} instead. Here's the right way to do the preceding broken code fragments:

```
for $i (1..10) {
    @array = somefunc($i);
    $AoA[$i] = [ @array ];
}
```

The square brackets make a reference to a new array with a *copy* of what's in @array at the time of the assignment. This is what you want.

Note that this will produce something similar, but it's much harder to read:

```
for $i (1..10) {
    @array = 0 .. $i;
    @{$AoA[$i]} = @array;
}
```

Is it the same? Well, maybe so–and maybe not. The subtle difference is that when you assign something in square brackets, you know for sure it's always a brand new reference with a new *copy* of the data. Something else could be going on in this new case with the `@{$AoA[$i]}` dereference on the left-hand-side of the assignment. It all depends on whether `$AoA[$i]` had been undefined to start with, or whether it already contained a reference. If you had already populated @AoA with references, as in

```
$AoA[3] = \@another_array;
```

Then the assignment with the indirection on the left-hand-side would use the existing reference that was already there:

```
@{$AoA[3]} = @array;
```

Of course, this *would* have the "interesting" effect of clobbering @another_array. (Have you ever noticed how when a programmer says something is "interesting", that rather than meaning "intriguing", they're disturbingly more apt to mean that it's "annoying", "difficult", or both? :-)

So just remember always to use the array or hash constructors with `[]` or {}, and you'll be fine, although it's not always optimally efficient.

Surprisingly, the following dangerous-looking construct will actually work out fine:

```
for $i (1..10) {
    my @array = somefunc($i);
    $AoA[$i] = \@array;
}
```

That's because my() is more of a run-time statement than it is a compile-time declaration *per se*. This means that the my() variable is remade afresh each time through the loop. So even though it *looks* as though you stored the same variable reference each time, you actually did not! This is a subtle distinction that can produce more efficient code at the risk of misleading all but the most experienced of programmers. So I usually advise against teaching it to beginners. In fact, except for passing arguments to functions, I seldom like to see the gimme-a-reference operator (backslash) used much at all in code. Instead, I advise beginners that they (and most of the rest of us) should try to use the much more easily understood constructors `[]` and {} instead of relying upon lexical (or dynamic) scoping and hidden reference-counting to do the right thing behind the scenes.

In summary:

```
$AoA[$i] = [ @array ];      # usually best
$AoA[$i] = \@array;         # perilous; just how my() was that array?
@{ $AoA[$i] } = @array;     # way too tricky for most programmers
```

7.5 CAVEAT ON PRECEDENCE

Speaking of things like `@{$AoA[$i]}`, the following are actually the same thing:

```
$aref->[2][2]       # clear
$$aref[2][2]        # confusing
```

That's because Perl's precedence rules on its five prefix dereferencers (which look like someone swearing: `$ @ * % &`) make them bind more tightly than the postfix subscripting brackets or braces! This will no doubt come as a great shock to the C or C++ programmer, who is quite accustomed to using `*a[i]` to mean what's pointed to by the *i'th* element of `a`. That is, they first take the subscript, and only then dereference the thing at that subscript. That's fine in C, but this isn't C.

The seemingly equivalent construct in Perl, `$$aref[$i]` first does the deref of $aref, making it take $aref as a reference to an array, and then dereference that, and finally tell you the *i'th* value of the array pointed to by $AoA. If you wanted the C notion, you'd have to write `${$AoA[$i]}` to force the `$AoA[$i]` to get evaluated first before the leading `$` dereferencer.

7.6 WHY YOU SHOULD ALWAYS `use strict`

If this is starting to sound scarier than it's worth, relax. Perl has some features to help you avoid its most common pitfalls. The best way to avoid getting confused is to start every program like this:

```
#!/usr/bin/perl -w
use strict;
```

This way, you'll be forced to declare all your variables with my() and also disallow accidental "symbolic dereferencing". Therefore if you'd done this:

```
my $aref = [
    [ "fred", "barney", "pebbles", "bambam", "dino", ],
    [ "homer", "bart", "marge", "maggie", ],
    [ "george", "jane", "elroy", "judy", ],
];

print $aref[2][2];
```

The compiler would immediately flag that as an error *at compile time*, because you were accidentally accessing `@aref`, an undeclared variable, and it would thereby remind you to write instead:

```
print $aref->[2][2]
```

7.7 DEBUGGING

Before version 5.002, the standard Perl debugger didn't do a very nice job of printing out complex data structures. With 5.002 or above, the debugger includes several new features, including command line editing as well as the `x` command to dump out complex data structures. For example, given the assignment to $AoA above, here's the debugger output:

```
DB<1> x $AoA
$AoA = ARRAY(0x13b5a0)
   0  ARRAY(0x1f0a24)
      0  'fred'
      1  'barney'
      2  'pebbles'
      3  'bambam'
      4  'dino'
   1  ARRAY(0x13b558)
      0  'homer'
      1  'bart'
      2  'marge'
      3  'maggie'
   2  ARRAY(0x13b540)
      0  'george'
      1  'jane'
      2  'elroy'
      3  'judy'
```

7.8 CODE EXAMPLES

Presented with little comment (these will get their own manpages someday) here are short code examples illustrating access of various types of data structures.

7.9 ARRAYS OF ARRAYS

Declaration of an ARRAY OF ARRAYS

```
@AoA = (
       [ "fred", "barney" ],
       [ "george", "jane", "elroy" ],
       [ "homer", "marge", "bart" ],
     );
```

Generation of an ARRAY OF ARRAYS

```
# reading from file
while ( <> ) {
    push @AoA, [ split ];
}

# calling a function
for $i ( 1 .. 10 ) {
    $AoA[$i] = [ somefunc($i) ];
}

# using temp vars
for $i ( 1 .. 10 ) {
    @tmp = somefunc($i);
    $AoA[$i] = [ @tmp ];
}

# add to an existing row
push @{ $AoA[0] }, "wilma", "betty";
```

Access and Printing of an ARRAY OF ARRAYS

```
# one element
$AoA[0][0] = "Fred";

# another element
$AoA[1][1] =~ s/(\w)/\u$1/;

# print the whole thing with refs
for $aref ( @AoA ) {
    print "\t [ @$aref ],\n";
}

# print the whole thing with indices
for $i ( 0 .. $#AoA ) {
    print "\t [ @{$AoA[$i]} ],\n";
}

# print the whole thing one at a time
for $i ( 0 .. $#AoA ) {
    for $j ( 0 .. $#{ $AoA[$i] } ) {
        print "elt $i $j is $AoA[$i][$j]\n";
    }
}
```

7.10 HASHES OF ARRAYS

Declaration of a HASH OF ARRAYS

```
%HoA = (
       flintstones        => [ "fred", "barney" ],
       jetsons            => [ "george", "jane", "elroy" ],
       simpsons           => [ "homer", "marge", "bart" ],
     );
```

Generation of a HASH OF ARRAYS

```
# reading from file
# flintstones: fred barney wilma dino
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $HoA{$1} = [ split ];
}

# reading from file; more temps
# flintstones: fred barney wilma dino
while ( $line = <> ) {
    ($who, $rest) = split /:\s*/, $line, 2;
    @fields = split ' ', $rest;
    $HoA{$who} = [ @fields ];
}

# calling a function that returns a list
for $group ( "simpsons", "jetsons", "flintstones" ) {
    $HoA{$group} = [ get_family($group) ];
}

# likewise, but using temps
for $group ( "simpsons", "jetsons", "flintstones" ) {
    @members = get_family($group);
    $HoA{$group} = [ @members ];
}
```

```
# append new members to an existing family
push @{ $HoA{"flintstones"} }, "wilma", "betty";
```

Access and Printing of a HASH OF ARRAYS

```
# one element
$HoA{flintstones}[0] = "Fred";

# another element
$HoA{simpsons}[1] =~ s/(\w)/\u$1/;

# print the whole thing
foreach $family ( keys %HoA ) {
    print "$family: @{ $HoA{$family} }\n"
}

# print the whole thing with indices
foreach $family ( keys %HoA ) {
    print "family: ";
    foreach $i ( 0 .. $#{ $HoA{$family} } ) {
        print " $i = $HoA{$family}[$i]";
    }
    print "\n";
}

# print the whole thing sorted by number of members
foreach $family ( sort { @{$HoA{$b}} <=> @{$HoA{$a}} } keys %HoA ) {
    print "$family: @{ $HoA{$family} }\n"
}

# print the whole thing sorted by number of members and name
foreach $family ( sort {
                            @{$HoA{$b}} <=> @{$HoA{$a}}
                                        ||
                                    $a cmp $b
           } keys %HoA )
{
    print "$family: ", join(", ", sort @{ $HoA{$family} }), "\n";
}
```

7.11 ARRAYS OF HASHES

Declaration of an ARRAY OF HASHES

```
@AoH = (
       {
           Lead     => "fred",
           Friend   => "barney",
       },
       {
           Lead     => "george",
           Wife     => "jane",
           Son      => "elroy",
       },
       {
           Lead     => "homer",
           Wife     => "marge",
           Son      => "bart",
       }
 );
```

Generation of an ARRAY OF HASHES

```
# reading from file
# format: LEAD=fred FRIEND=barney
while ( <> ) {
    $rec = {};
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $rec->{$key} = $value;
    }
    push @AoH, $rec;
}

# reading from file
# format: LEAD=fred FRIEND=barney
# no temp
while ( <> ) {
    push @AoH, { split /[\s+=]/ };
}

# calling a function  that returns a key/value pair list, like
# "lead","fred","daughter","pebbles"
while ( %fields = getnextpairset() ) {
    push @AoH, { %fields };
}

# likewise, but using no temp vars
while (<>) {
    push @AoH, { parsepairs($_) };
}

# add key/value to an element
$AoH[0]{pet} = "dino";
$AoH[2]{pet} = "santa's little helper";
```

Access and Printing of an ARRAY OF HASHES

```
# one element
$AoH[0]{lead} = "fred";

# another element
$AoH[1]{lead} =~ s/(\w)/\u$1/;

# print the whole thing with refs
for $href ( @AoH ) {
    print "{ ";
    for $role ( keys %$href ) {
        print "$role=$href->{$role} ";
    }
    print "}\n";
}

# print the whole thing with indices
for $i ( 0 .. $#AoH ) {
    print "$i is { ";
    for $role ( keys %{ $AoH[$i] } ) {
        print "$role=$AoH[$i]{$role} ";
    }
    print "}\n";
}
```

```
# print the whole thing one at a time
for $i ( 0 .. $#AoH ) {
    for $role ( keys %{ $AoH[$i] } ) {
        print "elt $i $role is $AoH[$i]{$role}\n";
    }
}
```

7.12 HASHES OF HASHES

Declaration of a HASH OF HASHES

```
%HoH = (
       flintstones => {
               lead      => "fred",
               pal       => "barney",
       },
       jetsons     => {
               lead      => "george",
               wife      => "jane",
               "his boy" => "elroy",
       },
       simpsons    => {
               lead      => "homer",
               wife      => "marge",
               kid       => "bart",
       },
);
```

Generation of a HASH OF HASHES

```
# reading from file
# flintstones: lead=fred pal=barney wife=wilma pet=dino
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $who = $1;
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $HoH{$who}{$key} = $value;
    }

# reading from file; more temps
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $who = $1;
    $rec = {};
    $HoH{$who} = $rec;
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $rec->{$key} = $value;
    }
}

# calling a function  that returns a key,value hash
for $group ( "simpsons", "jetsons", "flintstones" ) {
    $HoH{$group} = { get_family($group) };
}

# likewise, but using temps
for $group ( "simpsons", "jetsons", "flintstones" ) {
    %members = get_family($group);
```

```
    $HoH{$group} = { %members };
}

# append new members to an existing family
%new_folks = (
    wife => "wilma",
    pet  => "dino",
);

for $what (keys %new_folks) {
    $HoH{flintstones}{$what} = $new_folks{$what};
}
```

Access and Printing of a HASH OF HASHES

```
# one element
$HoH{flintstones}{wife} = "wilma";

# another element
$HoH{simpsons}{lead} =~ s/(\w)/\u$1/;

# print the whole thing
foreach $family ( keys %HoH ) {
    print "$family: { ";
    for $role ( keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}

# print the whole thing  somewhat sorted
foreach $family ( sort keys %HoH ) {
    print "$family: { ";
    for $role ( sort keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}

# print the whole thing sorted by number of members
foreach $family ( sort { keys %{$HoH{$b}} <=> keys %{$HoH{$a}} } keys %HoH ) {
    print "$family: { ";
    for $role ( sort keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}

# establish a sort order (rank) for each role
$i = 0;
for ( qw(lead wife son daughter pal pet) ) { $rank{$_} = ++$i }

# now print the whole thing sorted by number of members
foreach $family ( sort { keys %{ $HoH{$b} } <=> keys %{ $HoH{$a} } } keys %HoH ) {
    print "$family: { ";
    # and print these according to rank order
    for $role ( sort { $rank{$a} <=> $rank{$b} }  keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}
```

7.13 MORE ELABORATE RECORDS

Declaration of MORE ELABORATE RECORDS

Here's a sample showing how to create and use a record whose fields are of many different sorts:

```
$rec = {
    TEXT      => $string,
    SEQUENCE  => [ @old_values ],
    LOOKUP    => { %some_table },
    THATCODE  => \&some_function,
    THISCODE  => sub { $_[0] ** $_[1] },
    HANDLE    => \*STDOUT,
};

print $rec->{TEXT};

print $rec->{SEQUENCE}[0];
$last = pop @ { $rec->{SEQUENCE} };

print $rec->{LOOKUP}{"key"};
($first_k, $first_v) = each %{ $rec->{LOOKUP} };

$answer = $rec->{THATCODE}->($arg);
$answer = $rec->{THISCODE}->($arg1, $arg2);

# careful of extra block braces on fh ref
print { $rec->{HANDLE} } "a string\n";

use FileHandle;
$rec->{HANDLE}->autoflush(1);
$rec->{HANDLE}->print(" a string\n");
```

Declaration of a HASH OF COMPLEX RECORDS

```
%TV = (
   flintstones => {
       series   => "flintstones",
       nights   => [ qw(monday thursday friday) ],
       members  => [
           { name => "fred",    role => "lead", age  => 36, },
           { name => "wilma",   role => "wife", age  => 31, },
           { name => "pebbles", role => "kid",  age  =>  4, },
       ],
   },

   jetsons     => {
       series   => "jetsons",
       nights   => [ qw(wednesday saturday) ],
       members  => [
           { name => "george",  role => "lead", age  => 41, },
           { name => "jane",    role => "wife", age  => 39, },
           { name => "elroy",   role => "kid",  age  =>  9, },
       ],
    },

   simpsons    => {
       series   => "simpsons",
       nights   => [ qw(monday) ],
       members  => [
           { name => "homer", role => "lead", age  => 34, },
           { name => "marge", role => "wife", age => 37, },
```

```
            { name => "bart",  role => "kid",  age  =>  11, },
        ],
    },
);
```

Generation of a HASH OF COMPLEX RECORDS

```
# reading from file
# this is most easily done by having the file itself be
# in the raw data format as shown above.  perl is happy
# to parse complex data structures if declared as data, so
# sometimes it's easiest to do that

# here's a piece by piece build up
$rec = {};
$rec->{series} = "flintstones";
$rec->{nights} = [ find_days() ];

@members = ();
# assume this file in field=value syntax
while (<>) {
    %fields = split /[\s=]+/;
    push @members, { %fields };
}
$rec->{members} = [ @members ];

# now remember the whole thing
$TV{ $rec->{series} } = $rec;

###########################################################
# now, you might want to make interesting extra fields that
# include pointers back into the same data structure so if
# change one piece, it changes everywhere, like for example
# if you wanted a {kids} field that was a reference
# to an array of the kids' records without having duplicate
# records and thus update problems.
###########################################################
foreach $family (keys %TV) {
    $rec = $TV{$family}; # temp pointer
    @kids = ();
    for $person ( @{ $rec->{members} } ) {
        if ($person->{role} =~ /kid|son|daughter/) {
            push @kids, $person;
        }
    }
    # REMEMBER: $rec and $TV{$family} point to same data!!
    $rec->{kids} = [ @kids ];
}

# you copied the array, but the array itself contains pointers
# to uncopied objects. this means that if you make bart get
# older via

$TV{simpsons}{kids}[0]{age}++;

# then this would also change in
print $TV{simpsons}{members}[2]{age};

# because $TV{simpsons}{kids}[0] and $TV{simpsons}{members}[2]
# both point to the same underlying anonymous hash table
```

```
# print the whole thing
foreach $family ( keys %TV ) {
    print "the $family";
    print " is on during @{ $TV{$family}{nights} }\n";
    print "its members are:\n";
    for $who ( @{ $TV{$family}{members} } ) {
        print " $who->{name} ($who->{role}), age $who->{age}\n";
    }
    print "it turns out that $TV{$family}{lead} has ";
    print scalar ( @{ $TV{$family}{kids} } ), " kids named ";
    print join (", ", map { $_->{name} } @{ $TV{$family}{kids} } );
    print "\n";
}
```

7.14 Database Ties

You cannot easily tie a multilevel data structure (such as a hash of hashes) to a dbm file. The first problem is that all but GDBM and Berkeley DB have size limitations, but beyond that, you also have problems with how references are to be represented on disk. One experimental module that does partially attempt to address this need is the MLDBM module. Check your nearest CPAN site as described in *perlmodlib* for source code to MLDBM.

7.15 SEE ALSO

perlref(1), perllol(1), perldata(1), perlobj(1)

7.16 AUTHOR

Tom Christiansen <*tchrist@perl.com*>
Last update: Wed Oct 23 04:57:50 MET DST 1996

Index