NXLog Community Edition Reference Manual

NXLog Ltd.

Table of Contents

Chapter 1. Man Pages

1.1. nxlog(8)

NAME

nxlog - collects, processes, converts, and forwards event logs in many different formats

SYNOPSIS

nxlog [-c *conffile*] [-f]

nxlog [-c *conffile*] -v

nxlog [-r | -s]

DESCRIPTION

NXLog can process high volumes of event logs from many different sources. Supported types of log processing include rewriting, correlating, alerting, filtering, and pattern matching. Additional features include scheduling, log file rotation, buffering, and prioritized processing. After processing, NXLog can store or forward event logs in any of many supported formats. Inputs, outputs, and processing are implemented with a modular architecture and a powerful configuration language.

While the details provided here apply to NXLog installations on Linux and other UNIX-style operating systems in particular, a few Windows-specific notes are included.

OPTIONS

***-c** conffile*, ***--conf** conffile*
: Specify an alternate configuration file *conffile*. On Windows, this option must be used with **-f**. To change the configuration file used by the NXLog service on Windows, modify the service parameters.

-f**, **--foreground
: Run in foreground, do not daemonize.

-h**, **--help
: Print help.

-r**, **--reload
: Reload configuration of a running instance.

-s**, **--stop
: Send stop signal to a running instance.

-v**, **--verify
: Verify configuration file syntax.

SIGNALS

Various signals can be used to control the NXLog process. Some corresponding Windows control codes are also available; these are shown in parentheses where applicable.

SIGHUP

This signal causes NXLog to reload the configuration and restart the modules. On Windows, "sc stop nxlog" and "sc start nxlog" can be used instead.

SIGUSR1 (200)

This signal generates an internal log message with information about the current state of NXLog and its configured module instances. The message will be generated with INFO log level, written to the log file (if configured with LogFile), and available via the im_internal module.

SIGUSR2 (201)

This signal causes NXLog to switch to the DEBUG log level. This is equivalent to setting the LogLevel directive to `DEBUG` but does not require NXLog to be restarted.

SIGINT/SIGQUIT/SIGTERM

NXLog will exit if it receives one of these signals. On Windows, "sc stop nxlog" can be used instead.

On Linux/UNIX, a signal can be sent with the `kill` command. The following, for example, sends the SIGUSR1 signal:

```
kill -SIGUSR1 $(cat /var/run/nxlog/nxlog.pid)
```

On Windows, a signal can be sent with the `sc` command. The following, for example, sends the 200 signal:

```
sc control nxlog 200
```

FILES

/bin/nxlog

The main NXLog executable

/bin/nxlog-stmnt-verifier

This tool can be used to check NXLog Language statements. All statements are read from standard input and then validated. If a statement is invalid, the tool prints an error to standard error and exits non-zero.

/etc/nxlog.conf

The default configuration file

/usr/libexec/nxlog/modules/

The NXLog modules are located in this directory, by default. See the ModuleDir directive.

/var/spool/nxlog/configcache.dat

This is the position cache file where positions are saved. See the NoCache directive, in addition to CacheDir.

/var/run/nxlog/nxlog.pid

The process ID (PID) of the currently running NXLog process is written to this file. See the PidFile directive.

SEE ALSO

nxlog-processor(8)

NXLog website: https://nxlog.co

NXLog User Guide: https://nxlog.co/documentation/nxlog-user-guide

COPYRIGHT

1.2. nxlog-processor(8)

NAME

nxlog-processor - performs batch log processing

SYNOPSIS

nxlog-processor [-c *conffile*] [-v]

DESCRIPTION

The nxlog-processor tool is similar to the NXLog daemon and uses the same configuration file. However, it runs in the foreground and exits after all input log data has been processed. Common input sources are files and databases. This tool is useful for log processing tasks such as:

- loading a group of files into a database,
- converting between different formats,
- testing configuration, or
- doing offline event correlation.

While the details provided here apply to NXLog installations on Linux and other UNIX-style operating systems in particular, a few Windows-specific notes are included.

OPTIONS

***-c** conffile, **--conf** conffile*
  Specify an alternate configuration file *conffile*.

-h**, **--help
  Print help.

-v**, **--verify
  Verify configuration file syntax.

FILES

/bin/nxlog-processor
  The main NXLog-processor executable

/bin/nxlog-stmnt-verifier
  This tool can be used to check NXLog Language statements. All statements are read from standard input and then validated. If a statement is invalid, the tool prints an error to standard error and exits non-zero.

/etc/nxlog.conf
  The default configuration file

/var/spool/nxlog/configcache.dat
  This is the position cache file where positions are saved. To disable position caching, as may be desirable

when using nxlog-processor, set the NoCache directive to TRUE.

SEE ALSO

nxlog(8)

NXLog website: https://nxlog.co

NXLog User Guide: https://nxlog.co/documentation/nxlog-user-guide

COPYRIGHT

Chapter 2. Configuration

An NXLog configuration consists of global directives, module instances, and routes. The following sections list the core NXLog directives provided. Additional directives are provided at the module level. A valid configuration must contain at least one input module instance and at least one output module instance.

A module instance name may contain letters, digits, periods (`.`), and underscores (`_`). The first character in a module instance name must be a letter or an underscore. The corresponding regular expression is `[a-zA-Z_][a-zA-Z0-9._]*`.

A route instance name may contain letters, digits, periods (`.`), and underscores (`_`). The first character in a route instance name must be a letter, a digit, or an underscore. The corresponding regular expression is `[a-zA-Z0-9_][a-zA-Z0-9._]*`.

2.1. General Directives

The following directives can be used throughout the configuration file. These directives are handled by the configuration parser, and substitutions occur before the configuration check.

define

Use this directive to configure a constant or macro to be used later. Refer to a `define` by surrounding the name with percent signs (`%`). Enclose a group of statements with curly braces (`{}`).

Example 1. Using the define Directive

This configuration shows three example defines: `BASEDIR` is a constant, `IMPORTANT` is a statement, and `WARN_DROP` is a group of statements.

nxlog.conf

```
define BASEDIR /var/log
define IMPORTANT if $raw_event =~ /important/ \
                    $Message = 'IMPORTANT ' + $raw_event;
define WARN_DROP { log_warning("dropping message"); drop(); }

<Input messages>
    Module  im_file
    File    '%BASEDIR%/messages'
</Input>

<Input proftpd>
    Module  im_file
    File    '%BASEDIR%/proftpd.log'
    <Exec>
        %IMPORTANT%
        if $raw_event =~ /dropme/ %WARN_DROP%
    </Exec>
</Input>
```

include

This directive allows a specified file to be included in the current configuration file. Wildcarded filenames are supported.

NOTE	The SpoolDir directive only takes effect after the configuration is parsed, so relative paths specified with the **include** directive must be relative to the working directory NXLog was started from.

Example 2. Using the include Directive

This example includes a file relative to the directory NXLog is started from:

nxlog.conf

```
include modules/module1.conf
```

This example includes all matching files and uses an absolute path:

nxlog.conf

```
include /etc/nxlog.d/*.conf
```

2.2. Global Directives

CacheDir

This directive specifies a directory where the cache file (`configcache.dat`) should be written. This directive has a compiled-in value which is used by default.

FlowControl

This optional boolean directive specifies whether all input and processor modules should use flow control. This defaults to TRUE. See the description of the module level FlowControl directive for more information.

Group

Similar to User, NXLog will set the group ID to run under. The group can be specified by name or numeric ID. This directive has no effect when running on the Windows platform or with nxlog-processor(8).

IgnoreErrors

If set to FALSE, NXLog will stop when it encounters a problem with the configuration file (such as an invalid module directive) or if there is any other problem which would prevent all modules functioning correctly. If set to TRUE, NXLog will start after logging the problem. The default value is TRUE.

LogFile

NXLog will write its internal log to this file. If this directive is not specified, self logging is disabled. Note that the im_internal module can also be used to direct internal log messages to files or different output destinations, but this does not support log level below `INFO`. This **LogFile** directive is especially useful for debugging.

LogLevel

This directive has five possible values: `CRITICAL`, `ERROR`, `WARNING`, `INFO`, and `DEBUG`. It will set both the logging level used for LogFile and the standard output if NXLog is started in the foreground. The default **LogLevel** is `INFO`.

ModuleDir

By default the NXLog binaries have a compiled-in value for the directory to search for loadable modules. This can be overridden with this directive. The module directory contains sub-directories for each module type (extension, input, output, and processor), and the module binaries are located in those.

NoCache

Some modules save data to a cache file which is persisted across a shutdown/restart. Modules such as im_file will save the file position in order to continue reading from the same position after a restart as before. This caching mechanism can be explicitly turned off with this directive. This is mostly useful with nxlog-processor(8) in offline mode. If this boolean directive is not specified, it defaults to FALSE (caching is enabled). Note that many input modules, such as *im_file*, provide a SavePos directive that can be used to disable the position cache for a specific module instance. **SavePos** has no effect if the cache is disabled globally with `NoCache TRUE`.

NoFreeOnExit

This directive is for debugging. When set to TRUE, NXLog will not free module resources on exit, allowing valgrind to show proper stack trace locations in module function calls. The default value is FALSE.

Panic

A panic condition is a critical state which usually indicates a bug. Assertions are used in NXLog code for checking conditions where the code will not work unless the asserted condition is satisfied, and for security. Failing assertions result in a panic and suggest a bug in the code. A typical case is checking for NULL pointers before pointer dereference. This directive can take three different values: `HARD`, `SOFT`, or `OFF`. `HARD` will cause an abort in case the assertion fails. This is how most C based programs work. `SOFT` will cause an exception to be thrown at the place of the panic/assertion. In case of NULL pointer checks this is identical to a NullPointerException in Java. It is possible that NXLog can recover from exceptions and can continue to process log messages, or at least the other modules can. In case of assertion failure the location and the condition is printed at `CRITICAL` log level in `HARD` mode and `ERROR` log level in `SOFT` mode. If **Panic** is set to `OFF`, the failing condition is printed in the logs but the execution will continue on the normal code path. Most of the time this will result in a segmentation fault or other undefined behavior, though in some cases turning off a buggy assertion or panic will solve the problems caused by it in `HARD`/`SOFT` mode. The default value for **Panic** is `SOFT`.

PidFile

Under Unix operating systems, NXLog writes a PID file as other system daemons do. The default PID file can be overridden with this directive in case multiple daemon instances need to be running. This directive has no effect when running on the Windows platform or with nxlog-processor(8).

RootDir

NXLog will set its root directory to the value specified with this directive. If SpoolDir is also set, this will be relative to the value of **RootDir** (chroot() is called first). This directive has no effect when running on the Windows platform or with the nxlog-processor(8).

SpoolDir

NXLog will change its working directory to the value specified with this directive. This is useful with files created through relative filenames (for example, with om_file) and in case of core dumps. This directive has no effect with the nxlog-processor(8).

SuppressRepeatingLogs

Under some circumstances it is possible for NXLog to generate an extreme amount of internal logs consisting of the same message due to an incorrect configuration or a software bug. In this case, the LogFile can quickly consume the available disk space. With this directive, NXLog will write at most 2 lines per second if the same message is generated successively, by logging "last message repeated n times" messages. If this boolean directive is not specified, it defaults to TRUE (suppression of repeating messages is enabled).

Threads

This directive specifies the number of worker threads to use. The number of the worker threads is calculated and set to an optimal value if this directive is not defined. Do not set this unless you know what you are doing.

User

NXLog will drop to the user specified with this directive. This is useful if NXLog needs privileged access to some system resources (such as kernel messages or to bind a port below 1024). On Linux systems NXLog will use capabilities to access these resources. In this case NXLog must be started as root. The user can be specified by name or numeric ID. This directive has no effect when running on the Windows platform or with nxlog-processor(8).

2.3. Common Module Directives

The following directives are common to all modules. The Module directive is mandatory.

Module

This mandatory directive specifies which binary should be loaded. The module binary has a `.so` extension on Unix and a `.dll` on Windows platforms and resides under the ModuleDir location. Each module binary name is prefixed with `im_`, `pm_`, `om_`, or `xm_` (for *input*, *processor*, *output*, and *extension*, respectively). It is possible for multiple instances to use the same loadable binary. In this case the binary is only loaded once but instantiated multiple times. Different module instances may have different configurations.

FlowControl

This optional boolean directive specifies whether the module instance should use flow control. **FlowControl** is only valid for Input and Processor modules. By default, **FlowControl** is TRUE (enabled). This module-level directive can be used to override the global FlowControl directive.

When flow control is in effect, a module (Input or Processor) which tries to forward log data to the next module in the route will be suspended if the next module cannot accept more data. For example, if a network module (such as om_tcp) cannot forward logs because of a network error, the preceding module in the route will be paused. When flow control is disabled, the module will drop the log record if the queue of the next module in the route is full.

Disabling flow control can be useful when multiple output modules are configured to store or forward log data. When flow control is enabled, the output modules will only process log data if all outputs are functional. Consider the case where log data is stored in a file using om_file and also forwarded over the network using om_tcp. When flow control is enabled, a network disconnection will make the data flow stall and log data will not be written into the local file either. With flow control disabled, NXLog will write log data to the file and will drop messages that cannot be forwarded over the network.

WARNING	Suspending an im_udp instance is ineffective, because UDP provides no receipt acknowledgement. Suspending an im_uds instance when collecting local Syslog messages from the /dev/log Unix domain socket will cause the syslog() system call to block in any programs trying to write to the system log. It is generally recommended to disable flow control in these cases.

InputType

This directive specifies the name of the registered input reader function to be used for parsing raw events from input data. Names are treated case insensitively. This directive is only available for stream oriented input modules: im_file, im_exec, im_ssl, im_tcp, im_udp, and im_uds. These modules work by filling an input buffer with data read from the source. If the read operation was successful (there was data coming from the source), the module calls the specified callback function. If this is not explicitly specified, the module default will be used. Note that *im_udp* may only work properly if log messages do not span multiple packets and are within the UDP message size limit. Otherwise the loss of a packet may lead to parsing errors.

Modules may provide custom input reader functions. Once these are registered into the NXLog core, the modules listed above will be capable of using these. This makes it easier to implement custom protocols because these can be developed without concern for the transport layer.

The following input reader functions are provided by the NXLog core:

Binary

The input is parsed in the NXLog binary format, which preserves the parsed fields of the event records. The LineBased reader will automatically detect event records in the binary NXLog format, so it is only recommended to configure InputType to **Binary** if compatibility with other logging software is not required.

Dgram

Once the buffer is filled with data, it is considered to be one event record. This is the default for the im_udp input module, since UDP Syslog messages arrive in separate packets.

LineBased

The input is assumed to contain event records separated by newlines. It can handle both CRLF (Windows) and LF (Unix) line-breaks. Thus if an LF (`\n`) or CRLF (`\r\n`) is found, the function assumes that it has reached the end of the event record.

Example 3. TCP Input Assuming NXLog Format

This configuration explicitly specifies the Binary InputType.

nxlog.conf

```
<Input tcp>
    Module      im_tcp
    Port        2345
    InputType   Binary
</Input>
```

OutputType

This directive specifies the name of the registered output writer function to be used for formatting raw events when storing or forwarding output. Names are treated case insensitively. This directive is only available for stream oriented output modules: om_file, om_exec, om_ssl, om_tcp, om_udp, and om_uds. These modules work by filling the output buffer with data to be written to the destination. The specified callback function is called before the write operation. If this is not explicitly specified, the module default will be used.

Modules may provide custom output formatter functions. Once these are registered into the NXLog core, the modules listed above will be capable of using these. This makes it easier to implement custom protocols because these can be developed without concern for the transport layer.

The following output writer functions are provided by the NXLog core:

Binary

The output is written in the NXLog binary format which preserves parsed fields of the event records.

Dgram

Once the buffer is filled with data, it is considered to be one event record. This is the default for the om_udp output module, since UDP Syslog messages are sent in separate packets.

LineBased

The output will contain event records separated by newlines. The record terminator is CRLF (`\r\n`).

Example 4. TCP Output Sending Messages in NXLog Format

This configuration explicitly specifies the Binary OutputType.

nxlog.conf

```
<Output tcp>
    Module      om_tcp
    Port        2345
    Host        localhost
    OutputType  Binary
</Output>
```

2.3.1. Exec

The **Exec** directive/block contains statements in the NXLog language which are executed when a module receives a log message. This directive is available in all input, processor, and output modules. It is not available in most extension modules because these do not handle log messages directly (the xm_multiline and xm_rewrite modules do provide **Exec** directives).

Example 5. Simple Exec Statement

This statement assigns a value to the `$Hostname` field in the event record.

nxlog.conf

```
Exec    $Hostname = 'myhost';
```

Each directive must be on one line unless it contains a trailing backslash (`\`) character.

Example 6. Exec Statement Spanning Multiple Lines

This if statement uses line continuation to span multiple lines.

nxlog.conf

```
Exec    if $Message =~ /something interesting/       \
            log_info("found something interesting"); \
        else                                         \
            log_debug("found nothing interesting");
```

More than one **Exec** directive or block may be specified. They are executed in the order of appearance. Each **Exec** directive must contain a full statement. Therefore it is not possible to split the lines in the previous example into multiple **Exec** directives. It is only possible to split the **Exec** directive if it contains multiple statements.

Example 7. Equivalent Use of Statements in Exec

This example shows two equivalent uses of the **Exec** directive.

nxlog.conf

```
Exec    log_info("first"); \
        log_info("second");
```

This produces identical behavior:

nxlog.conf

```
Exec    log_info("first");
Exec    log_info("second");
```

The **Exec** directive can also be used as a block. To use multiple statements spanning more than one line, it is recommended to use the `<Exec>` block instead. When using a block, it is not necessary to use the backslash (`\`) character for line continuation.

Example 8. Using the Exec Block

This example shows two equivalent uses of **Exec**, first as a *directive*, then as a *block*.

nxlog.conf

```
Exec    log_info("first"); \
        log_info("second");
```

The following **Exec** *block* is equivalent. Notice the backslash (`\`) is omitted.

nxlog.conf

```
<Exec>
    log_info("first");
    log_info("second");
</Exec>
```

2.3.2. Schedule

The Schedule block can be used to execute periodic jobs, such as log rotation or any other task. Scheduled jobs have the same priority as the module. The Schedule block has the following directives:

Every

In addition to the crontab format it is possible to schedule execution at periodic intervals. With the crontab format it is not possible to run a job every five days for example, but this directive enables it in a simple way. It takes an integer value with an optional unit. The unit can be one of the following: `sec`, `min`, `hour`, `day`, or `week`. If the unit is not specified, the value is assumed to be in seconds.

Exec

The mandatory **Exec** directive takes one or more NXLog statements. This is the code which is actually being scheduled. Multiple **Exec** directives can be specified within one **Schedule** block. See the module-level Exec directive, this behaves the same. Note that it is not possible to use fields in statements here because execution is not triggered by log messages.

First

This directive sets the first execution time. If the value is in the past, the next execution time is calculated as if NXLog has been running since and jobs will not be run to make up for missed events in the past. The directive takes a datetime literal value.

When

This directive takes a value similar to a crontab entry: five space-separated definitions for minute, hour, day, month, and weekday. See the crontab(5) manual for the field definitions. It supports lists as comma separated values and/or ranges. Step values are also supported with the slash. Month and week days are not supported, these must be defined with numeric values. The following extensions are also supported:

```
@yearly        Run once a year, "0 0 1 1 *".
@annually      (same as @yearly)
@monthly       Run once a month, "0 0 1 * *".
@weekly        Run once a week, "0 0 * * 0".
@daily         Run once a day, "0 0 * * *".
@midnight      (same as @daily)
@hourly        Run once an hour, "0 * * * *".
```

Example 9. Scheduled Exec Statements

This example shows two scheduled Exec statements in a im_tcp module instance. The first is executed every second, while the second uses a crontab(5) style value.

nxlog.conf

```
<Input in>
    Module  im_tcp
    Port    2345

    <Schedule>
        Every   1 sec
        First   2010-12-17 00:19:06
        Exec    log_info("scheduled execution at " + now());
    </Schedule>

    <Schedule>
        When    1 */2 2-4 * *
        Exec    log_info("scheduled execution at " + now());
    </Schedule>
</Input>
```

2.4. Route Directives

The following directives can be used in Route blocks. The Path directive is mandatory.

Path

The data flow is defined by the **Path** directive. First the instance names of Input modules are specified. If more than one Input reads log messages which feed data into the route, then these must be separated by commas. The list of Input modules is followed by an arrow (`=>`). Either processor modules or output modules follow. Processor modules must be separated by arrows, not commas, because they operate in series, unlike Input and Output modules which work in parallel. Output modules are separated by commas. The **Path** must specify at least an Input and an Output. The syntax is illustrated by the following:

```
Path INPUT1[, INPUT2...] => [PROCESSOR1 [=> PROCESSOR2...] =>] OUTPUT1[, OUTPUT2...]
```

Example 10. Specifying Routes

The following configuration shows modules being used in three different routes.

nxlog.conf

```
<Input in1>
    Module  im_null
</Input>

<Input in2>
    Module  im_null
</Input>

<Processor p1>
    Module  pm_null
</Processor>

<Processor p2>
    Module  pm_null
</Processor>

<Output out1>
    Module  om_null
</Output>

<Output out2>
    Module  om_null
</Output>

<Route 1>
    # Basic route
    Path    in1 => out1
</Route>

<Route 2>
    # Basic route with one processor module
    Path    in1 => p1 => out1
</Route>

<Route 3>
    # Complex route with multiple input/output/processor modules
    Path    in1, in2 => p1 => p2 => out1, out2
</Route>
```

Priority

This directive takes an integer value in the range of 1-100 as a parameter, and the default is `10`. Log messages in routes with a lower **Priority** value will be processed before others. Internally, this value is assigned to each module part of the route. The events of the modules are processed in priority order by the NXLog engine. Modules of a route with a lower **Priority** value (higher priority) will process log messages first.

Example 11. Prioritized Processing

This configuration prioritizes the UDP route over the TCP route in order to minimize loss of UDP Syslog messages when the system is busy.

nxlog.conf

```
<Input tcpin>
    Module      im_tcp
    Host        localhost
    Port        514
</Input>

<Input udpin>
    Module      im_udp
    Host        localhost
    Port        514
</Input>

<Output tcpfile>
    Module      om_file
    File        "/var/log/tcp.log"
</Output>

<Output udpfile>
    Module      om_file
    File        "/var/log/udp.log"
</Output>

<Route udp>
    Priority    1
    Path        udpin => udpfile
</Route>

<Route tcp>
    Priority    2
    Path        tcpin => tcpfile
</Route>
```

Chapter 3. Language

3.1. Types

The following types are provided by the NXLog language.

Unknown
: This is a special type for values where the type cannot be determined at compile time and for uninitialized values. The undef literal and fields without a value also have an unknown type. The unknown type can also be thought of as "any" in case of function and procedure API declarations.

Boolean
: A boolean value is TRUE, FALSE or undefined. Note that an undefined value is not the same as a FALSE value.

Integer
: An integer can hold a signed 64 bit value in addition to the undefined value. Floating point values are not supported.

String
: A string is an array of characters in any character set. The binary type should be used for values where the NUL byte can also occur. An undefined string is not the same as an empty string. Strings have a limited length to prevent resource exhaustion problems, this is a compile-time value currently set to 1M.

Datetime
: A datetime holds a microsecond value of time elapsed since the Epoch. It is always stored in UTC/GMT.

IPv4 Address
: An ip4addr type stores a dotted-quad IPv4 address in an internal format (integer).

IPv6 Address
: An ip6addr type stores an IPv6 address in an internal format.

Regular expression
: A regular expression type can only be used with the =~ or !~ operators.

Binary
: This type can hold an array of bytes.

Variadic arguments
: This is a special type only used in function and procedure API declarations to indicate variadic arguments.

3.2. Expressions

3.2.1. Literals

Undef
: The undef literal has an unknown type. It can be also used in an assignment to unset the value of a field.

Example 12. Un-Setting the Value of a Field

This statement unsets the `$ProcessID` field.

```
$ProcessID = undef;
```

Boolean

A boolean literal is either TRUE or FALSE. It is case-insensitive, so `True`, `False`, `true`, and `false` are also valid.

Integer

An integer starts with a minus (`-`) sign if it is negative. A "0X" or "0x" prepended modifier indicates a hexadecimal notation. The "K", "M" and "G" modifiers are also supported; these mean Kilo (1024), Mega (1024^2), or Giga (1024^3) respectively when appended.

Example 13. Setting an Integer Value

This statement uses a modifier to set the `$Limit` field to 44040192 (42×1024^2).

```
$Limit = 42M;
```

String

String literals are quoted characters using either single or double quotes. String literals specified with double quotes can contain the following escape sequences.

\\

The backslash (`\`) character.

\\"

The double quote (`"`) character.

\n

Line feed (LF).

\r

Carriage return (CR).

\t

Horizontal tab.

\b

Audible bell.

\xXX

A single byte in the form of a two digit hexadecimal number. For example the line-feed character can also be expressed as `\x0A`.

NOTE	String literals in single quotes do not process the escape sequences: `"\n"` is a single character (LF) while `'\n'` is two characters. The following comparison is FALSE for this reason: `"\n" == '\n'`.

NOTE	Extra care should be taken with the backslash when using double quoted string literals to specify file paths on Windows. For more information about the possible complications, see this note for the *im_file* **File** directive.

Example 14. Setting a String Value

This statement sets the `$Message` field to the specified string.

```
$Message = "Test message";
```

Regular expression

Regular expressions must be quoted with slashes as in Perl. Captured substrings are accessible through a numeric reference such as $1. The full subject string is placed into $0.

Example 15. A regular expression match operation

```
if $Message =~ /^Test (\S+)/ log_info("captured: " + $1);
```

Datetime

A datetime literal is an unquoted representation of a time value expressing local time in the format of `YYYY-MM-DD hh:mm:ss`.

Example 16. Setting a Datetime Value

This statement sets the `$EventTime` field to the specified datetime value.

```
$EventTime = 2000-01-02 03:04:05;
```

IPv4 Address

An IPv4 literal value is expressed in dotted quad notation such as `192.168.1.1`.

IPv6 Address

An IPv6 literal value is expressed by 8 groups of 16-bit hexadecimal values separated by colons (`:`) such as `2001:0db8:85a3:0000:0000:8a2e:0370:7334`.

3.2.2. Fields

Fields are referenced in the NXLog language by prepending a dollar sign (`$`) to the field name.

Normally, a field name may contain letters, digits, the period (`.`), and the underscore (`_`). Additionally, field names must begin with a letter or an underscore. The corresponding regular expression is:

```
[a-zA-Z_][a-zA-Z0-9._]*
```

However, those restrictions are relaxed if the field name is specified with curly braces (`{}`). In this case, the field name may also contain hyphens (`-`), parentheses (`()`), and spaces. The field name may also begin with any one of the allowed characters. The regular expression in this case is:

```
[a-zA-Z0-9._() -]+
```

Example 17. Referencing a Field

This statement generates an internal log message indicating the time when the message was received by NXLog.

```
log_debug('Message received at ' + $EventReceivedTime);
```

This statement uses curly braces (`{}`) to refer to a field with a hyphenated name.

```
log_info('The file size is ' + ${file-size});
```

A field which does not exist has an unknown type.

3.2.3. Operations

3.2.3.1. Unary Operations

The following unary operations are available. It is possible to use brackets around the operand to make it look like a function call as in the "defined" example below.

not

The **not** operator expects a boolean value. It will evaluate to undef if the value is undefined. If it receives an unknown value which evaluates to a non-boolean, it will result in a run-time execution error.

Example 18. Using the "not" Operand

If the `$Success` field has a value of false, an error is logged.

```
if not $Success log_error("Job failed");
```

defined

The defined operator will evaluate to TRUE if the operand is defined, otherwise FALSE.

Example 19. Using the Unary "defined" Operation

This statement is a no-op, it does nothing.

```
if defined undef log_info("never printed");
```

If the `$EventTime` field has not been set (due perhaps to failed parsing), it will be set to the current time.

```
if not defined($EventTime) $EventTime = now();
```

3.2.3.2. Binary Operations

The following binary operations are available.

The operations are described with the following syntax:

`LEFT_OPERAND_TYPE OPERATION RIGHT_OPERAND_TYPE = EVALUATED_VALUE_TYPE`

=~

This is the regular expression match operation as in Perl. The PCRE engine is used to to execute the regular expressions. This operation takes a string and a regexp operand and evaluates to a boolean value which will be TRUE if the regular expression matches the subject string. Captured sub-strings are accessible through numeric reference, such as `$1`, and the full subject string is placed into `$0`.

- `string =~ regexp = boolean`
- `regexp =~ string = boolean`

Example 20. Regular Expression Based String Matching

A log message will be generated if the `$Message` field matches the regular expression.

```
if $Message =~ /^Test message/ log_info("matched");
```

Regular expression based string substitution is also supported with the `s///` operator.

The following regular expression modifiers are supported:

g

The `/g` modifier can be used for global replacement.

Example 21. Replace Whitespace Occurrences

```
if $SourceName =~ s/\s/_/g log_info("removed all whitespace in SourceName");
```

s

The dot (`.`) normally matches any character except newline. The `/s` modifier causes the dot to match all characters including line terminator characters (LF and CRLF).

Example 22. Dot Matches All Characters

```
if $Message =~ /^Backtrace.*END$/s drop();
```

m

The `/m` modifier can be used to treat the string as multiple lines (`^` and `$` match newlines within data).

i

The `/i` modifier does case insensitive matching.

Variables and captured sub-string references cannot be used inside the regular expression or the regexp substitution operator (they will be treated literally).

!~

This is the opposite of `=~`: the expression will evaluate to TRUE if the regular expression does not match on the subject string. It can be also written as `not LEFT_OPERAND =~ RIGHT_OPERAND`.

- `string !~ regexp = boolean`
- `regexp !~ string = boolean`

The `s///` substitution operator is also supported.

Example 23. Regular Expression Based Negative String Matching

A log message will be generated if the `$Message` field does not match the regular expression.

```
if $Message !~ /^Test message/ log_info("didn't match");
```

==

This operator compares two values for equality. Comparing a defined value with an undefined results in undef.

- `undef == undef = TRUE`
- `string == string = boolean`
- `integer == integer = boolean`
- `boolean == boolean = boolean`

- `datetime == datetime = boolean`
- `ip4addr == ip4addr = boolean`
- `ip4addr == string = boolean`
- `string == ip4addr = boolean`

Example 24. Equality

A log message will be generated if `$SeverityValue` is 1.

```
if $SeverityValue == 1 log_info("severity is one");
```

!=

This operator compares two values for inequality. Comparing a defined value with an undefined results in undef.

- `undef != undef = FALSE`
- `string != string = boolean`
- `integer != integer = boolean`
- `boolean != boolean = boolean`
- `datetime != datetime = boolean`
- `ip4addr != ip4addr = boolean`
- `ip4addr != string = boolean`
- `string != ip4addr = boolean`

Example 25. Inequality

A log message will be generated if `$SeverityValue` is not 1.

```
if $SeverityValue != 1 log_info("severity is not one");
```

<

This operation will evaluate to TRUE if the left operand is less than the right operand, and FALSE otherwise. Comparing a defined value with an undefined results in undef.

- `integer < integer = boolean`
- `datetime < datetime = boolean`

Example 26. Less

A log message will be generated if `$SeverityValue` is less than 1.

```
if $SeverityValue < 1 log_info("severity is less than one");
```

<=

This operation will evaluate to TRUE if the left operand is less than or equal to the right operand, and FALSE otherwise. Comparing a defined value with an undefined results in undef.

- `integer <= integer = boolean`
- `datetime <= datetime = boolean`

Example 27. Less or Equal

A log message will be generated if `$SeverityValue` is less than or equal to 1.

```
if $SeverityValue < 1 log_info("severity is less than or equal to one");
```

>

This operation will evaluate to TRUE if the left operand is greater than the right operand, and FALSE otherwise. Comparing a defined value with an undefined results in undef.

- `integer > integer = boolean`
- `datetime > datetime = boolean`

Example 28. Greater

A log message will be generated if `$SeverityValue` is greater than 1.

```
if $SeverityValue > 1 log_info("severity is greater than one");
```

>=

This operation will evaluate to TRUE if the left operand is greater than or equal to the right operand, and FALSE otherwise. Comparing a defined value with an undefined results in undef.

- `integer >= integer = boolean`
- `datetime >= datetime = boolean`

Example 29. Greater or Equal

A log message will be generated if `$SeverityValue` is greater than or equal to 1.

```
if $SeverityValue >= 1 log_info("severity is greater than or equal to one");
```

and

This operation evaluates to TRUE if and only if both operands are TRUE. The operation will evaluate to undef if either operand is undefined.

`boolean and boolean = boolean`

Example 30. And Operation

A log message will be generated only if both `$SeverityValue` equals 1 *and* `$FacilityValue` equals 2.

```
if $SeverityValue == 1 and $FacilityValue == 2 log_info("1 and 2");
```

or

This operation evaluates to TRUE if either operand is TRUE. The operation will evaluate to undef if both operands are undefined.

`boolean or boolean = boolean`

Example 31. Or Operation

A log message will be generated if `$SeverityValue` is equal to either 1 or 2.

```
if $SeverityValue == 1 or $SeverityValue == 2 log_info("1 or 2");
```

+

This operation will result in an integer if both operands are integers. If either operand is a string, the result will be a string where non-string typed values are converted to strings. In this case it acts as a concatenation operator, like the dot (`.`) operator in Perl. Adding an undefined value to a non-string will result in undef.

- `integer + integer = integer`
- `string + undef = string`
- `undef + string = string`
- `undef + undef = undef`
- `string + string = string` (Concatenate two strings.)
- `datetime + integer = datetime` (Add the number of seconds in the right value to the datetime stored in the left value.)
- `integer + datetime = datetime` (Add the number of seconds in the left value to the datetime stored in the right value.)

Example 32. Concatenation

This statement will always cause a log message to be generated.

```
if 1 + "a" == "1a" log_info("this will be printed");
```

-

Subtraction. The result will be undef if either operand is undefined.

- `integer - integer = integer` (Subtract two integers.)
- `datetime - datetime = integer` (Subtract two datetime types. The result is the difference between to two expressed in microseconds.)
- `datetime - integer = datetime` (Subtract the number of seconds from the datetime stored in the left value.)

Example 33. Subtraction

This statement will always cause a log message to be generated.

```
if 4 - 1 == 3 log_info("four minus one is three");
```

*

Multiply an integer with another. The result will be undef if either operand is undefined.

`integer * integer = integer`

Example 34. Multiplication

This statement will always cause a log message to be generated.

```
if 4 * 2 == 8 log_info("four times two is eight");
```

/

Divide an integer with another. The result will be undef if either operand is undefined. Since the result is an integer, a fractional part is lost.

`integer / integer = integer`

Example 35. Division

This statement will always cause a log message to be generated.

```
if 9 / 4 == 2 log_info("9 divided by 4 is 2");
```

%

The modulo operation divides an integer with another and returns the remainder. The result will be undef if either operand is undefined.

`integer % integer = integer`

Example 36. Modulo

This statement will always cause a log message to be generated.

```
if 3 % 2 == 1 log_info("three mod two is one");
```

IN

This operation will evaluate to TRUE if the left operand is equal to any of the expressions in the list on the right, and FALSE otherwise. Comparing a undefined value results in undef.

`unknown IN unknown, unknown … = boolean`

Example 37. IN

A log message will be generated if `$EventID` is equal to any one of the values in the list.

```
if $EventID IN (1000, 1001, 1004, 4001) log_info("EventID found");
```

NOT IN

This operation is equivalent to `NOT expr IN expr_list`.

`unknown NOT IN unknown, unknown … = boolean`

Example 38. NOT IN

A log message will be generated if `$EventID` is not equal to any of the values in the list.

```
if $EventID NOT IN (1000, 1001, 1004, 4001) log_info("EventID not in list");
```

3.2.4. Functions

See Functions for a list of functions provided by the NXLog core. Additional functions are available through modules.

Example 39. A Function Call

This statement uses the now() function to set the field to the current time.

```
$EventTime = now();
```

It is also possible to call a function of a specific module instance.

Example 40. Calling a Function of a Specific Module Instance

This statement calls the file_name() and file_size() functions of a defined *om_file* instance named `out` in order to log the name and size of its currently open output file.

```
log_info('Size of output file ' + out->file_name() + ' is ' + out->file_size());
```

3.3. Statements

The following elements can be used in statements. There is no loop operation (*for* or *while*) in the NXLog language.

3.3.1. Assignment

The assignment operation is declared with an equal sign (`=`). It loads the value from the expression evaluated on the right into a field on the left.

Example 41. Field Assignment

This statement sets the `$EventReceivedTime` field to the value returned by the now() function.

```
$EventReceivedTime = now();
```

3.3.2. Block

A block consists of one or more statements within curly braces (`{}`). This is typically used with conditional statements as in the example below.

Example 42. Conditional Statement Block

If the expression matches, both log messages will be generated.

```
if now() > 2000-01-01 00:00:00
{
   log_info("we are in the");
   log_info("21st century");
}
```

3.3.3. Procedures

See Procedures for a list of procedures provided by the NXLog core. Additional procedures are available through modules.

Example 43. A Procedure Call

The log_info() procedure generates an internal log message.

```
log_info("No log source activity detected.");
```

It is also possible to call a procedure of a specific module instance.

Example 44. Calling a Procedure of a Specific Module Instance

This statement calls the parse_csv() procedure of a defined *xm_csv* module instance named `csv_parser`.

```
csv_parser->parse_csv();
```

3.3.4. If-Else

A conditional statement starts with the `if` keyword followed by a boolean expression and a statement. The `else` keyword, followed by another statement, is optional. Brackets around the expression are also optional.

Example 45. Conditional Statements

A log message will be generated if the expression matches.

```
if now() > 2000-01-01 00:00:00 log_info("we are in the 21st century");
```

This statement is the same as the previous, but uses brackets.

```
if ( now() > 2000-01-01 00:00:00 ) log_info("we are in the 21st century");
```

This is a conditional statement block.

```
if now() > 2000-01-01 00:00:00
{
   log_info("we are in the 21st century");
}
```

This conditional statement block includes an else branch.

```
if now() > 2000-01-01 00:00:00
{
   log_info("we are in the 21st century");
}
else log_info("we are not yet in the 21st century");
```

Like Perl, the NXLog language does not have a *switch* statement. Instead, this can be accomplished by using conditional *if-else* statements.

Example 46. Emulating switch with if-else

The generated log message various based on the value of the `$value` field.

```
if ( $value == 1 )
   log_info("1");
else if ( $value == 2 )
   log_info("2");
else if ( $value == 3 )
   log_info("3");
else
   log_info("default");
```

NOTE	The Perl *elsif* and *unless* keywords are not supported.

3.4. Variables

A module variable can only be accessed from the same module instance where it was created. A variable is referenced by a string value and can store a value of any type.

See the create_var(), delete_var(), set_var(), and get_var() procedures.

3.5. Statistical Counters

The following types are available for statistical counters:

COUNT
: Added values are aggregated, and the value of the counter is increased if only positive integers are added until the counter is destroyed or indefinitely if the counter has no expiry.

COUNTMIN
: This calculates the minimum value of the counter.

COUNTMAX
: This calculates the maximum value of the counter.

AVG
: This algorithm calculates the average over the specified interval.

AVGMIN
: This algorithm calculates the average over the specified interval, and the value of the counter is always the lowest which was ever calculated during the lifetime of the counter.

AVGMAX
: Like AVGMIN, but this returns the highest value calculated during the lifetime of the counter.

RATE
: This calculates the value over the specified interval. It can be used to calculate events per second (EPS) values.

RATEMIN
: This calculates the value over the specified interval, and returns the lowest rate calculated during the lifetime of the counter.

RATEMAX

Like RATEMIN, but this returns the highest rate calculated during the lifetime of the counter.

GRAD

This calculates the change of the rate of the counter over the specified interval, which is the gradient.

GRADMIN

This calculates the gradient and returns the lowest gradient calculated during the lifetime of the counter.

GRADMAX

Like GRADMIN, but this returns the highest gradient calculated during the lifetime of the counter.

Example 47. Simple Event Correlation Using Statistical Counters

If the number of login failures exceeds 3 within 45 seconds, then an internal log message is generated. This accomplishes the exact same task as our previous algorithm did with module variables, except that it is a lot simpler. In addition, this method is more precise, because it uses the timestamp from the log message instead of relying on the current time; consequently it is also possible to use this for offline log analysis.

```
if $Message =~ /login failure/
{
   # create will not do anything if the counter already exists
   create_stat('login_failures', 'RATE', 45, $EventTime);
   add_stat('login_failures', 1, $EventTime);
   if get_stat('login_failures', $EventTime) >= 3
      log_warning(">= 3 login failures detected within 45 seconds");
}
```

Note that this is still not perfect because the time window used in the rate calculation does not shift, so the problem described in our previous example also affects this version, and this algorithm may not work in some situations. For this reason and for better performance, it is better to use the event correlation module instead; it has a Thresholded rule which uses a sliding window to overcome this problem.

3.6. Functions

The following functions are exported by *core*.

datetime `datetime(integer arg)`

Convert the integer argument, expressing the number of microseconds since epoch, to datetime.

integer `day(datetime datetime)`

Return the day part of the time value.

integer `dayofweek(datetime datetime)`

Return the number of days since Sunday in the range of 0-6.

integer `dayofyear(datetime datetime)`

Return the day number of the year in the range of 1-366.

boolean `dropped()`

Return TRUE if the currently processed event has already been dropped.

datetime `fix_year(datetime datetime)`

Set the year value to the current year in a *datetime* which was parsed with a missing year, such as BSD Syslog or Cisco timestamps.

integer `get_stat(string statname)`

Return the value of the statistical counter or undef if it does not exist.

integer `get_stat(string statname, datetime time)`

Return the value of the statistical counter or undef if it does not exist. The *time* argument specifies the current time.

unknown `get_var(string varname)`

Return the value of the variable or undef if it does not exist.

ip4addr `host_ip()`

Return the first non-loopback IP address the hostname resolves to.

ip4addr `host_ip(integer nth)`

Return the *nth* non-loopback IP address the hostname resolves to. The *nth* argument starts from 1.

string `hostname()`

Return the hostname (short form).

string `hostname_fqdn()`

Return the FQDN hostname. This function will return the short form if the FQDN hostname cannot be determined.

integer `hour(datetime datetime)`

Return the hour part of the time value.

integer `integer(unknown arg)`

Parse and convert the string argument to an integer. For datetime type it returns the number of microseconds since epoch.

ip4addr `ip4addr(integer arg)`

Convert the integer argument to an ip4addr type.

ip4addr `ip4addr(integer arg, boolean ntoa)`

Convert the integer argument to an ip4addr type. If *ntoa* is set to true, the integer is assumed to be in network byte order. Instead of `1.2.3.4` the result will be `4.3.2.1`.

string `lc(string arg)`

Convert the string to lower case.

integer `microsecond(datetime datetime)`

Return the microsecond part of the time value.

integer `minute(datetime datetime)`

Return the minute part of the time value.

integer `month(datetime datetime)`

Return the month part of the *datetime* value.

datetime `now()`

Return the current time.

datetime `parsedate(string arg)`

Parse a string containing a timestamp. Dates without timezone information are treated as local time. The current year is used for formats that do not include the year. An undefined datetime type is returned if the argument cannot be parsed, so that the user can fix the error (for example, `$EventTime = parsedate($somestring); if not defined($EventTime) $EventTime = now();`). Supported timestamp formats are listed below.

RFC 3164 (legacy Syslog) and variations

```
Nov 6 08:49:37
Nov  6 08:49:37
Nov 06 08:49:37
Nov 3 14:50:30.403
Nov  3 14:50:30.403
Nov 03 14:50:30.403
Nov 3 2005 14:50:30
Nov  3 2005 14:50:30
Nov 03 2005 14:50:30
Nov 3 2005 14:50:30.403
Nov  3 2005 14:50:30.403
Nov 03 2005 14:50:30.403
```

RFC 1123

RFC 1123 compliant dates are also supported, including a couple others which are similar such as those defined in RFC 822, RFC 850, and RFC 1036.

```
Sun, 06 Nov 1994 08:49:37 GMT  ; RFC 822, updated by RFC 1123
Sunday, 06-Nov-94 08:49:37 GMT ; RFC 850, obsoleted by RFC 1036
Sun Nov  6 08:49:37 1994       ; ANSI C's asctime() format
Sun, 6 Nov 1994 08:49:37 GMT   ; RFC 822, updated by RFC 1123
Sun, 06 Nov 94 08:49:37 GMT    ; RFC 822
Sun,  6 Nov 94 08:49:37 GMT    ; RFC 822
Sun, 6 Nov 94 08:49:37 GMT     ; RFC 822
Sun, 06 Nov 94 08:49 GMT       ; Unknown
Sun, 6 Nov 94 08:49 GMT        ; Unknown
Sun, 06 Nov 94 8:49:37 GMT     ; Unknown [Elm 70.85]
Sun, 6 Nov 94 8:49:37 GMT      ; Unknown [Elm 70.85]
Mon,  7 Jan 2002 07:21:22 GMT  ; Unknown [Postfix]
Sun, 06-Nov-1994 08:49:37 GMT  ; RFC 850 with four digit years
```

The above formats are also recognized when the leading day of week and/or the timezone are omitted.

Apache/NCSA date

This format can be found in Apache access logs and other sources.

```
24/Aug/2009:16:08:57 +0200
```

ISO 8601 and RFC 3339

NXLog can parse the ISO format with or without sub-second resolution, and with or without timezone information. It accepts either a comma (`,`) or a dot (`.`) in case there is sub-second resolution.

```
1977-09-06 01:02:03
1977-09-06 01:02:03.004
1977-09-06T01:02:03.004Z
1977-09-06T01:02:03.004+02:00
2011-5-29 0:3:21
2011-5-29 0:3:21+02:00
2011-5-29 0:3:21.004
2011-5-29 0:3:21.004+02:00
```

Windows timestamp

This format is `YYYYMMDDhhmmss.USEC` with an optional timezone offset.

```
20100426151354.537875-000
20100426151354.537875000
```

Integer timestamp

This format is `XXXXXXXXXX.USEC`. The value is expressed as an integer showing the number of seconds elapsed since the epoch UTC. The fractional microsecond part is optional.

```
1258531221.650359
1258531221
```

string `replace(string subject, string src, string dst)`

Replace all occurrences of *src* with *dst* in the *subject* string.

string `replace(string subject, string src, string dst, integer count)`

Replace *count* number occurrences of *src* with *dst* in the *subject* string.

integer `second(datetime datetime)`

Return the second part of the time value.

integer `size(string str)`

Return the size of the string *str* in bytes.

string `strftime(datetime datetime, string fmt)`

Convert a datetime to a string with the given format. See the strftime(3) manual or the Windows strftime reference for the format specification.

string `string(unknown arg)`

Convert the argument to a string.

datetime `strptime(string input, string fmt)`

Convert the string to a datetime with the given format. See the manual of strptime(3) for the format specification.

string `substr(string src, integer from)`

Return the string starting at the byte offset specified in *from*.

string `substr(string src, integer from, integer to)`

Return a sub-string specified with the starting and ending positions as byte offsets from the beginning of the string.

string `type(unknown arg)`

Return the type of the variable, which can be `boolean`, `integer`, `string`, `datetime`, `ip4addr`, `ip6addr`, `regexp`, or `binary`. For values with the unknown type, it returns undef.

string `uc(string arg)`

Convert the string to upper case.

integer `year(datetime datetime)`

Return the year part of the *datetime* value.

3.7. Procedures

The following procedures are exported by *core*.

`add_stat(string statname, integer value);`

Add *value* to the statistical counter using the current time.

`add_stat(string statname, integer value, datetime time);`

Add *value* to the statistical counter using the time specified in the argument named *time*.

`add_to_route(string routename);`

Copy the currently processed event data to the route specified. This procedure makes a copy of the data. The original will be processed normally. Note that flow control is explicitly disabled when moving data with add_to_route() and the data will not be added if the queue of the target module(s) is full.

`create_stat(string statname, string type);`

Create a module statistical counter with the specified name using the current time. The statistical counter will be created with an infinite lifetime. The *type* argument must be one of the following to select the required algorithm for calculating the value of the statistical counter: `COUNT`, `COUNTMIN`, `COUNTMAX`, `AVG`, `AVGMIN`, `AVGMAX`, `RATE`, `RATEMIN`, `RATEMAX`, `GRAD`, `GRADMIN`, or `GRADMAX` (see Statistical Counters).

This procedure with two parameters can only be used with `COUNT`, otherwise the interval parameter must be specified (see below). This procedure will do nothing if a counter with the specified name already exists.

`create_stat(string statname, string type, integer interval);`

Create a module statistical counter with the specified name to be calculated over *interval* seconds and using the current time. The statistical counter will be created with an infinite lifetime.

`create_stat(string statname, string type, integer interval, datetime time);`

Create a module statistical counter with the specified name to be calculated over *interval* seconds and the time value specified in the *time* argument. The statistical counter will be created with an infinite lifetime.

`create_stat(string statname, string type, integer interval, datetime time, integer lifetime);`

Create a module statistical counter with the specified name to be calculated over *interval* seconds and the time value specified in the *time* argument. The statistical counter will expire after *lifetime* seconds.

`create_stat(string statname, string type, integer interval, datetime time, datetime expiry);`

Create a module statistical counter with the specified name to be calculated over *interval* seconds and the time value specified in the *time* argument. The statistical counter will expire at *expiry*.

`create_var(string varname);`

Create a module variable with the specified name. The variable will be created with an infinite lifetime.

`create_var(string varname, integer lifetime);`

Create a module variable with the specified name and the *lifetime* given in seconds. When the lifetime expires, the variable will be deleted automatically and `get_var(name)` will return undef.

`create_var(string varname, datetime expiry);`

Create a module variable with the specified name. The *expiry* specifies when the variable should be deleted automatically.

`debug(unknown arg, varargs args);`

Print the argument(s) at DEBUG log level. Same as log_debug().

`delete(unknown arg);`

Delete the field from the event. For example, `delete($field)`. Note that `$field = undef` is not the same, though after both operations the field will be undefined.

`delete_var(string varname);`

Delete the module variable with the specified name if it exists.

`drop();`

Drop the event record that is currently being processed. Any further action on the event record will result in a "missing logdata" error.

`log_debug(unknown arg, varargs args);`

Print the argument(s) at DEBUG log level. Same as debug().

`log_error(unknown arg, varargs args);`

Print the argument(s) at ERROR log level.

`log_info(unknown arg, varargs args);`

Print the argument(s) at INFO log level.

`log_warning(unknown arg, varargs args);`

Print the argument(s) at WARNING log level.

`rename_field(string old, string new);`

Rename a field. For example, `rename_field("old", "new")`.

`reroute(string routename);`

Move the currently processed event data to the route specified. The event data will enter the route as if it was received by an input module there. Note that flow control is explicitly disabled when moving data with reroute() and the data will be dropped if the queue of the target module(s) is full.

`set_var(string varname, unknown value);`

Set the value of a module variable. If the variable does not exist, it will be created with an infinite lifetime.

`sleep(integer interval);`

Sleep the specified number of microseconds. This procedure is provided for testing purposes primarily. It can be used as a poor man's rate limiting tool, though this use is not recommended.

Chapter 4. Extension Modules

Extension modules do not process log messages directly, and for this reason their instances cannot be part of a route. These modules enhance the features of NXLog in various ways, such as exporting new functions and procedures or registering additional I/O reader and writer functions (to be used with modules supporting the InputType and OutputType directives). There are many ways to hook an extension module into the NXLog engine, as the following modules illustrate.

4.1. Character Set Conversion (xm_charconv)

This module provides tools for converting strings between different character sets (codepages). All the encodings available to *iconv* are supported. See `iconv -l` for a list of encoding names.

4.1.1. Configuration

The *xm_charconv* module accepts the following directives in addition to the common module directives.

AutodetectCharsets

This optional directive accepts a comma-separated list of character set names. When `auto` is specified as the source encoding for convert() or convert_fields(), these character sets will be tried for conversion.

4.1.2. Functions

The following functions are exported by *xm_charconv*.

string `convert(string source, string srcencoding, string dstencoding)`

Convert the source string to the encoding specified in *dstencoding* from *srcencoding*. The *srcencoding* argument can be set to `auto` to request auto detection.

4.1.3. Procedures

The following procedures are exported by *xm_charconv*.

`convert_fields(string srcencoding, string dstencoding);`

Convert all string type fields of a log message from *srcencoding* to *dstencoding*. The *srcencoding* argument can be set to `auto` to request auto detection.

4.1.4. Examples

Example 48. Character set auto-detection of various input encodings

This configuration shows an example of character set auto-detection. The input file can contain differently encoded lines, and the module normalizes output to UTF-8.

nxlog.conf

```
<Extension charconv>
    Module              xm_charconv
    AutodetectCharsets  utf-8, euc-jp, utf-16, utf-32, iso8859-2
</Extension>

<Input filein>
    Module              im_file
    File                "tmp/input"
    Exec                convert_fields("auto", "utf-8");
</Input>

<Output fileout>
    Module              om_file
    File                "tmp/output"
</Output>

<Route r>
    Path                filein => fileout
</Route>
```

4.2. Delimiter-Separated Values (xm_csv)

This module provides functions and procedures for working with data formatted as comma-separated values (CSV). CSV input can be parsed into fields and CSV output can be generated. Delimiters other than the comma can be used also.

The pm_transformer module provides a simple interface to parse and generate CSV format, but the *xm_csv* module exports an API that can be used to solve more complex tasks involving CSV formatted data.

NOTE	It is possible to use more than one *xm_csv* module instance with different options in order to support different CSV formats at the same time. For this reason, functions and procedures exported by the module are public and must be referenced by the module instance name.

4.2.1. Configuration

The *xm_csv* module accepts the following directives in addition to the common module directives. The Fields directive is required.

Fields

This mandatory directive accepts a comma-separated list of fields which will be filled from the input parsed. Field names with or without the dollar sign (`$`) are accepted. The fields will be stored as strings unless their types are explicitly specified with the FieldTypes directive.

Delimiter

This optional directive takes a single character (see below) as argument to specify the delimiter character used to separate fields. The default delimiter character is the comma (`,`). Note that there is no delimiter after the last field.

EscapeChar

This optional directive takes a single character (see below) as argument to specify the escape character used to escape special characters. The escape character is used to prefix the following characters: the escape character itself, the quote character, and the delimiter character. If EscapeControl is TRUE, the newline (`\n`), carriage return (`\r`), tab (`\t`), and backspace (`\b`) control characters are also escaped. The default escape character is the backslash character (`\`).

EscapeControl

If this optional boolean directive is set to TRUE, control characters are also escaped. See the EscapeChar directive for details. The default is TRUE: control characters are escaped. Note that this is necessary to allow single line CSV field lists which contain line-breaks.

FieldTypes

This optional directive specifies the list of types corresponding to the field names defined in Fields. If specified, the number of types must match the number of field names specified with Fields. If this directive is omitted, all fields will be stored as strings. This directive has no effect on the fields-to-CSV conversion.

QuoteChar

This optional directive takes a single character (see below) as argument to specify the quote character used to enclose fields. If QuoteOptional is TRUE, then only string type fields are quoted. The default is the double-quote character (`"`).

QuoteMethod

This optional directive can take the following values:

All

All fields will be quoted.

None

Nothing will be quoted. This can be problematic if a field value (typically text that can contain any character) contains the delimiter character. Make sure that this is escaped or replaced with something else.

String

Only string type fields will be quoted. This has the same effect as QuoteOptional set to TRUE and is the default behavior if the **QuoteMethod** directive is not specified.

Note that this directive only effects CSV generation when using to_csv(). The CSV parser can automatically detect the quotation.

QuoteOptional

This directive has been deprecated in favor of QuoteMethod, which should be used instead.

UndefValue

This optional directive specifies a string which will be treated as an undefined value. This is particularly useful when parsing the W3C format where the dash (`-`) marks an omitted field.

4.2.1.1. Specifying Quote, Escape, and Delimiter Characters

The QuoteChar, EscapeChar, and Delimiter directives can be specified in several ways.

Unquoted single character

Any printable character can be specified as an unquoted character, except for the backslash (`\`):

```
Delimiter ;
```

Control characters

The following non-printable characters can be specified with escape sequences:

\a

audible alert (bell)

\b

backspace

\t

horizontal tab

\n

newline

\v

vertical tab

\f

formfeed

\r

carriage return

For example, to use TAB delimiting:

```
Delimiter \t
```

A character in single quotes

The configuration parser strips whitespace, so it is not possible to define a space as the delimiter unless it is enclosed within quotes:

```
Delimiter ' '
```

Printable characters can also be enclosed:

```
Delimiter ';'
```

The backslash can be specified when enclosed within quotes:

```
Delimiter '\'
```

A character in double quotes

Double quotes can be used like single quotes:

```
Delimiter " "
```

The backslash can be specified when enclosed within double quotes:

```
Delimiter "\"
```

A hexadecimal ASCII code

Hexadecimal ASCII character codes can also be used by prepending `0x`. For example, the space can be specified as:

```
Delimiter 0x20
```

This is equivalent to:

```
Delimiter " "
```

4.2.2. Functions

The following functions are exported by *xm_csv*.

string *to_csv()*

Convert the specified fields to a single CSV formatted string.

4.2.3. Procedures

The following procedures are exported by *xm_csv*.

parse_csv();

Parse the `$raw_event` field as CSV input.

parse_csv(string source);

Parse the given string as CSV format.

to_csv();

Format the specified fields as CSV and put this into the `$raw_event` field.

4.2.4. Examples

Example 49. Complex CSV Format Conversion

This example shows that the *xm_csv* module can not only parse and create CSV formatted input and output, but with multiple *xm_csv* modules it is also possible to reorder, add, remove, or modify fields before outputting to a different CSV format.

nxlog.conf

```
<Extension csv1>
    Module      xm_csv
    Fields      $id, $name, $number
    FieldTypes  integer, string, integer
    Delimiter   ,
</Extension>

<Extension csv2>
    Module      xm_csv
    Fields      $id, $number, $name, $date
    Delimiter   ;
</Extension>

<Input in>
    Module      im_file
    File        "tmp/input"
    <Exec>
        csv1->parse_csv();
        $date = now();
        if not defined $number $number = 0;
        csv2->to_csv();
    </Exec>
</Input>

<Output out>
    Module      om_file
    File        "tmp/output"
</Output>
```

Input Sample

```
1, "John K.", 42
2, "Joe F.", 43
```

Output Sample

```
1;42;"John K.";2011-01-15 23:45:20
2;43;"Joe F.";2011-01-15 23:45:20
```

4.3. External Programs (xm_exec)

This module provides two procedures which make it possible to execute external scripts or programs. These two procedures are provided through this extension module in order to keep the NXLog core small. Also, without this module loaded an administrator is not able to execute arbitrary scripts.

NOTE	The im_exec and om_exec modules also provide support for running external programs, though the purpose of these is to pipe data to and read data from programs. The procedures provided by the *xm_exec* module do not pipe log message data, but are intended for multiple invocations (though data can be still passed to the executed script/program as command line arguments).

4.3.1. Configuration

The *xm_exec* module accepts only the common module directives.

4.3.2. Procedures

The following procedures are exported by *xm_exec*.

`exec(string command, varargs args);`

Execute *command*, passing it the supplied arguments, and wait for it to terminate. The command is executed in the caller module's context. Note that the module calling this procedure will block until the process terminates. Use the exec_async() procedure to avoid this problem. All output written to standard output and standard error by the spawned process is discarded.

`exec_async(string command, varargs args);`

This procedure executes the command passing it the supplied arguments and does not wait for it to terminate.

4.3.3. Examples

Example 50. NXLog Acting as a Cron Daemon

This *xm_exec* module instance will run the command every second without waiting for it to terminate.

nxlog.conf

```
<Extension exec>
    Module  xm_exec
    <Schedule>
        Every   1 sec
        Exec    exec_async("/bin/true");
    </Schedule>
</Extension>
```

Example 51. Sending Email Alerts

If the `$raw_event` field matches the regular expression, an email will be sent.

nxlog.conf

```
<Extension exec>
    Module  xm_exec
</Extension>

<Input tcp>
    Module  im_tcp
    Host    0.0.0.0
    Port    1514
    <Exec>
      if $raw_event =~ /alertcondition/
      {
      exec_async("/bin/sh", "-c", 'echo "' + $Hostname +
        '\n\nRawEvent:\n' + $raw_event +
        '"|/usr/bin/mail -a "Content-Type: text/plain; charset=UTF-8" -s "ALERT" ' +
        'user@domain.com');
      }
    </Exec>
</Input>

<Output file>
    Module  om_file
    File    "/var/log/messages"
</Output>

<Route tcp_to_file>
    Path    tcp => file
</Route>
```

For another example, see File Rotation Based on Size.

4.4. File Operations (xm_fileop)

This module provides functions and procedures to manipulate files. Coupled with a Schedule block, this module allows various log rotation and retention policies to be implemented, including:

- log file retention based on file size,
- log file retention based on file age, and
- cyclic log file rotation and retention.

NOTE	Rotating, renaming, or removing the file written by om_file is also supported with the help of the *om_file* reopen() procedure.

4.4.1. Configuration

The *xm_fileop* module accepts only the common module directives.

4.4.2. Functions

The following functions are exported by *xm_fileop*.

boolean `dir_exists(string path)`

Return TRUE if *path* exists and is a directory. On error undef is returned and an error is logged.

string `dir_temp_get()`

Return the name of a directory suitable as a temporary storage location.

string `file_basename(string file)`

Strip the directory name from the full *file* path. For example, `basename('/var/log/app.log')` will return `app.log`.

datetime `file_ctime(string file)`

Return the creation or inode-changed time of *file*. On error undef is returned and an error is logged.

string `file_dirname(string file)`

Return the directory name of the full *file* path. For example, `basename('/var/log/app.log')` will return `/var/log`. Returns an empty string if *file* does not contain any directory separators.

boolean `file_exists(string file)`

Return TRUE if *file* exists and is a regular file.

integer `file_inode(string file)`

Return the inode number of *file*. On error undef is returned and an error is logged.

datetime `file_mtime(string file)`

Return the last modification time of *file*. On error undef is returned and an error is logged.

string `file_read(string file)`

Return the contents of *file* as a string value. On error undef is returned and an error is logged.

integer `file_size(string file)`

Return the size of *file*, in bytes. On error undef is returned and an error is logged.

string `file_type(string file)`

Return the type of *file*. The following string values can be returned: FILE, DIR, CHAR, BLOCK, PIPE, LINK, SOCKET, and UNKNOWN. On error undef is returned and an error is logged.

4.4.3. Procedures

The following procedures are exported by *xm_fileop*.

`dir_make(string path);`

Create a directory recursively (like `mkdir -p`). It succeeds if the directory already exists. An error is logged if the operation fails.

`dir_remove(string file);`

Remove the directory from the filesystem.

`file_append(string src, string dst);`

Append the contents of the file *src* to *dst*. The *dst* file will be created if it does not exist. An error is logged if the operation fails.

`file_chmod(string file, integer mode);`

Change the permissions of *file*. This function is only implemented on POSIX systems where chmod() is available in the underlying operating system. An error is logged if the operation fails.

`file_chown(string file, integer uid, integer gid);`

Change the ownership of *file*. This function is only implemented on POSIX systems where chown() is available in the underlying operating system. An error is logged if the operation fails.

`file_chown(string file, string user, string group);`

Change the ownership of *file*. This function is only implemented on POSIX systems where chown() is available in the underlying operating system. An error is logged if the operation fails.

`file_copy(string src, string dst);`

Copy the file *src* to *dst*. If file *dst* already exists, its contents will be overwritten. An error is logged if the operation fails.

`file_cycle(string file);`

Do a cyclic rotation on *file*. The *file* will be moved to "*file*.1". If "*file*.1" already exists it will be moved to "*file*.2", and so on. This procedure will reopen the LogFile if it is cycled. An error is logged if the operation fails.

`file_cycle(string file, integer max);`

Do a cyclic rotation on *file*. The *file* will be moved to "*file*.1". If "*file*.1" already exists it will be moved to "*file*.2", and so on. The *max* argument specifies the maximum number of files to keep. For example, if *max* is `5`, "*file*.6" will be deleted. This procedure will reopen the LogFile if it is cycled. An error is logged if the operation fails.

`file_link(string src, string dst);`

Create a hardlink from *src* to *dst*. An error is logged if the operation fails.

`file_remove(string file);`

Remove *file*. It is possible to specify a wildcard in the filename (but not in the path). The backslash (`\`) must be escaped if used as the directory separator with wildcards (for example, `C:\\test\\*.log`). This procedure will reopen the LogFile if it is removed. An error is logged if the operation fails.

`file_remove(string file, datetime older);`

Remove *file* if its creation time is older than the value specified in *older*. It is possible to specify a wildcard in the filename (but not in the path). The backslash (`\`) must be escaped if used as the directory separator with wildcards (for example, `C:\\test\\*.log`). This procedure will reopen the LogFile if it is removed. An error is logged if the operation fails.

`file_rename(string old, string new);`

Rename the file *old* to *new*. If the file *new* exists, it will be overwritten. Moving files or directories across devices may not be possible. This procedure will reopen the LogFile if it is renamed. An error is logged if the operation fails.

`file_touch(string file);`

Update the last modification time of *file* or create the *file* if it does not exist. An error is logged if the operation fails.

`file_truncate(string file);`

Truncate *file* to zero length. If the *file* does not exist, it will be created. An error is logged if the operation fails.

`file_truncate(string file, integer offset);`

Truncate *file* to the size specified in *offset*. If the *file* does not exist, it will be created. An error is logged if the operation fails.

file_write(string file, string value);
Write *value* into *file*. The *file* will be created if it does not exist. An error is logged if the operation fails.

4.4.4. Examples

Example 52. Rotation of the Internal LogFile

In this example, the internal log file is rotated based on time and size.

nxlog.conf

```
#define LOGFILE C:\Program Files (x86)\nxlog\data\nxlog.log
define LOGFILE /var/log/nxlog/nxlog.log

<Extension fileop>
    Module      xm_fileop

    # Check the log file size every hour and rotate if larger than 1 MB
    <Schedule>
        Every   1 hour
        Exec    if (file_size('%LOGFILE%') >= 1M) file_cycle('%LOGFILE%', 2);
    </Schedule>

    # Rotate log file every week on Sunday at midnight
    <Schedule>
        When    @weekly
        Exec    file_cycle('%LOGFILE%', 2);
    </Schedule>
</Extension>
```

4.5. GELF (xm_gelf)

This module provides an output writer function which can be used to generate output in Graylog Extended Log Format (GELF) for Graylog2 or GELF compliant tools.

Unlike Syslog format (with Snare Agent, for example), the GELF format contains structured data in JSON so that the fields are available for analysis. This is especially convenient with sources such as the Windows EventLog which already generate logs in a structured format.

The *xm_gelf* module provides the following output writer functions:

OutputType GELF_TCP
This output writer generates GELF for use with TCP (use with the om_tcp output module).

OutputType GELF_UDP
This output writer generates GELF for use with UDP (use with the om_udp output module).

OutputType GELF
This type is equivalent to `GELF_UDP`.

The GELF output generated by this module includes all fields, except for the $raw_event field and any field having a leading dot (`.`) or underscore (`_`).

Configure NXLog to output GELF formatted data by following these steps:

1. Load the *xm_gelf* module:

```
<Extension _gelf>
    Module      xm_gelf
</Extension>
```

2. Set the OutputType to `GELF_UDP` in the om_udp output module:

```
<Output out_udp>
    Module      om_udp
    Host        127.0.0.1
    Port        12201
    OutputType  GELF_UDP
</Output>
```

Or, for om_tcp, use `GELF_TCP`:

```
<Output out_tcp>
    Module      om_tcp
    Host        127.0.0.1
    Port        12201
    OutputType  GELF_TCP
</Output>
```

4.5.1. Configuration

The *xm_gelf* module accepts the following directives in addition to the common module directives.

ShortMessageLength

This optional directive can be used to specify the length of the *short_message* field. This defaults to 64 if the directive is not explicitly specified. If the field *short_message* or *ShortMessage* is present, it will not be truncated.

UseNullDelimiter

If this optional boolean directive is TRUE, `GELF_TCP` will use the NUL delimiter. If this directive is FALSE, it will use the newline delimiter. The default is TRUE.

4.5.2. Examples

Example 53. Sending Windows EventLog to Graylog2 in GELF

The following configuration reads the Windows EventLog and sends it to a Graylog2 server in GELF format.

nxlog.conf

```
<Extension gelf>
    Module      xm_gelf
</Extension>

<Input eventlog>
    # Use 'im_mseventlog' for Windows XP, 2000 and 2003
    Module      im_msvistalog
    # Uncomment the following to collect specific event logs only
    #Query  <QueryList>\
    #           <Query Id="0">\
    #               <Select Path="Application">*</Select>\
    #               <Select Path="System">*</Select>\
    #               <Select Path="Security">*</Select>\
    #           </Query>\
    #       </QueryList>
</Input>

<Output udp>
    Module      om_udp
    Host        192.168.1.1
    Port        12201
    OutputType  GELF
</Output>

<Route eventlog_to_udp>
    Path        eventlog => udp
</Route>
```

Example 54. Forwarding Custom Log Files to Graylog2 in GELF

In this example, custom application logs are collected and sent out in GELF, with custom fields set to make the data more useful for the receiver.

nxlog.conf

```
<Extension gelf>
    Module      xm_gelf
</Extension>

<Input file>
    Module      im_file
    File        "/var/log/app*.log"

    <Exec>
        # Set the $EventTime field usually found in the logs by
        # extracting it with a regexp. If this is not set, the current
        # system time will be used which might be a little off.
        if $raw_event =~ /(\d\d\d\d\-\d\d-\d\d \d\d:\d\d:\d\d)/
            $EventTime = parsedate($1);

        # Explicitly set the Hostname. This defaults to the system's
        # hostname if unset.
        $Hostname = 'myhost';

        # Now set the severity level to something custom. This defaults
        # to 'INFO' if unset. We can use the following numeric values
        # here which are the standard Syslog values: ALERT: 1, CRITICAL:
        # 2, ERROR: 3, WARNING: 4, NOTICE: 5, INFO: 6, DEBUG: 7
        if $raw_event =~ /ERROR/ $SyslogSeverityValue = 3;
        else $SyslogSeverityValue = 6;

        # Set a field to contain the name of the source file
        $FileName = file_name();

        # To set a custom message, use the $Message field. The
        # $raw_event field is used if $Message is unset.
        if $raw_event =~ /something important/
            $Message = 'IMPORTANT!! ' + $raw_event;
    </Exec>
</Input>

<Output udp>
    Module      om_udp
    Host        192.168.1.1
    Port        12201
    OutputType  GELF
</Output>

<Route file_to_gelf>
    Path        file => udp
</Route>
```

Example 55. Parsing a CSV File and Sending it to Graylog2 in GELF

With this configuration, NXLog will read a CSV file containing three fields and forward the data in GELF so that the fields will be available on the server.

nxlog.conf

```
<Extension gelf>
    Module      xm_gelf
</Extension>

<Extension csv>
    Module      xm_csv
    Fields      $name, $number, $location
    FieldTypes  string, integer, string
    Delimiter   ,
</Extension>

<Input file>
    Module      im_file
    File        "/var/log/app/csv.log"
    Exec        csv->parse_csv();
</Input>

<Output udp>
    Module      om_udp
    Host        192.168.1.1
    Port        12201
    OutputType  GELF
</Output>

<Route csv_to_gelf>
    Path        file => udp
</Route>
```

4.6. JSON (xm_json)

This module provides functions and procedures for processing data formatted as JSON. JSON can be generated from log data, or JSON can be parsed into fields. Unfortunately, the JSON specification does not define a type for datetime values so these are represented as JSON strings. The JSON parser in *xm_json* can automatically detect datetime values, so it is not necessary to explicitly use parsedate().

4.6.1. Configuration

The *xm_json* module accepts only the common module directives.

4.6.2. Functions

The following functions are exported by *xm_json*.

string `to_json()`

Convert the fields to JSON and return this as a string value. The `$raw_event` field and any field having a leading dot (`.`) or underscore (`_`) will be automatically excluded.

4.6.3. Procedures

The following procedures are exported by *xm_json*.

`parse_json();`

Parse the `$raw_event` field as JSON input.

`parse_json(string source);`

Parse the given string as JSON format.

`to_json();`

Convert the fields to JSON and put this into the `$raw_event` field. The `$raw_event` field and any field having a leading dot (`.`) or underscore (`_`) will be automatically excluded.

4.6.4. Examples

Example 56. Syslog to JSON Format Conversion

The following configuration accepts Syslog (both BSD and IETF) via TCP and converts it to JSON.

nxlog.conf

```
<Extension syslog>
    Module  xm_syslog
</Extension>

<Extension json>
    Module  xm_json
</Extension>

<Input tcp>
    Module  im_tcp
    Port    1514
    Host    0.0.0.0
    Exec    parse_syslog(); to_json();
</Input>

<Output file>
    Module  om_file
    File    "/var/log/json.txt"
</Output>

<Route tcp_to_file>
    Path    tcp => file
</Route>
```

Input Sample

```
<30>Sep 30 15:45:43 host44.localdomain.hu acpid: 1 client rule loaded
```

Output Sample

```
{
  "MessageSourceAddress":"127.0.0.1",
  "EventReceivedTime":"2011-03-08 14:22:41",
  "SyslogFacilityValue":1,
  "SyslogFacility":"DAEMON",
  "SyslogSeverityValue":5,
  "SyslogSeverity":"INFO",
  "SeverityValue":2,
  "Severity":"INFO",
  "Hostname":"host44.localdomain.hu",
  "EventTime":"2011-09-30 14:45:43",
  "SourceName":"acpid",
  "Message":"1 client rule loaded "
}
```

Example 57. Converting Windows EventLog to Syslog-Encapsulated JSON

The following configuration reads the Windows EventLog and converts it to the BSD Syslog format, with the message part containing the fields in JSON.

nxlog.conf

```
<Extension syslog>
    Module      xm_syslog
</Extension>

<Extension json>
    Module      xm_json
</Extension>

<Input eventlog>
    Module      im_msvistalog
    Exec        $Message = to_json(); to_syslog_bsd();
</Input>

<Output tcp>
    Module      om_tcp
    Host        192.168.1.1
    Port        1514
</Output>

<Route eventlog_json_tcp>
    Path        eventlog => tcp
</Route>
```

Output Sample

```
<14>Mar  8 14:40:11 WIN-OUNNPISDHIG Service_Control_Manager: {"EventTime":"2012-03-08
14:40:11","EventTimeWritten":"2012-03-08 14:40:11","Hostname":"WIN-
OUNNPISDHIG","EventType":"INFO","SeverityValue":2,"Severity":"INFO","SourceName":"Service
Control
Manager","FileName":"System","EventID":7036,"CategoryNumber":0,"RecordNumber":6788,"Message":"T
he nxlog service entered the running state. ","EventReceivedTime":"2012-03-08 14:40:12"}
```

4.7. Key-Value Pairs (xm_kvp)

This module provides functions and procedures for processing data formatted as key-value pairs (KVPs), also commonly called "name-value pairs". The module can both parse and generate key-value formatted data.

It is quite common to have a different set of keys in each log line when accepting key-value formatted input messages. Extracting values from such logs using regular expressions can be quite cumbersome. The *xm_kvp* extension module automates this process.

Log messages containing key-value pairs typically look like one the following:

- `key1: value1, key2: value2, key42: value42`
- `key1="value 1"; key2="value 2"`
- `Application=smtp, Event='Protocol Conversation', status='Client Request', ClientRequest='HELO 1.2.3.4'`

Keys are usually separated from the value using an equal sign (`=`) or a colon (`:`); and the key-value pairs are delimited with a comma (`,`), a semicolon (`;`), or a space. In addition, values and keys may be quoted and may contain escaping. The module will try to guess the format, or the format can be explicitly specified using the configuration directives below.

NOTE	It is possible to use more than one *xm_kvp* module instance with different options in order to support different KVP formats at the same time. For this reason, functions and procedures exported by the module are public and must be referenced by the module instance name.

4.7.1. Configuration

The *xm_kvp* module accepts the following directives in addition to the common module directives.

EscapeChar

This optional directive takes a single character (see below) as argument. It specifies the character used for escaping special characters. The escape character is used to prefix the following characters: the **EscapeChar** itself, the KeyQuoteChar, and the ValueQuoteChar. If EscapeControl is TRUE, the newline (`\n`), carriage return (`\r`), tab (`\t`), and backspace (`\b`) control characters are also escaped. The default escape character is the backslash (`\`).

EscapeControl

If this optional boolean directive is set to TRUE, control characters are also escaped. See the EscapeChar directive for details. The default is TRUE (control characters are escaped). Note that this is necessary in order to support single-line KVP field lists containing line-breaks.

KeyQuoteChar

This optional directive takes a single character (see below) as argument. It specifies the quote character for enclosing key names. If this directive is not specified, the module will accept single-quoted keys, double-quoted keys, and unquoted keys.

KVDelimiter

This optional directive takes a single character (see below) as argument. It specifies the delimiter character used to separate the key from the value. If this directive is not set and the parse_kvp() procedure is used, the module will try to guess the delimiter from the following: the colon (`:`) or the equal-sign (`=`).

KVPDelimiter

This optional directive takes a single character (see below) as argument. It specifies the delimiter character used to separate the key-value pairs. If this directive is not set and the parse_kvp() procedure is used, the module will try to guess the delimiter from the following: the comma (`,`), the semicolon (`;`), or the space.

ValueQuoteChar

This optional directive takes a single character (see below) as argument. It specifies the quote character for enclosing key values. If this directive is not specified, the module will accept single-quoted values, double-quoted values, and unquoted values. Normally, quotation is used when the value contains a space or the KVDelimiter character.

4.7.1.1. Specifying Quote, Escape, and Delimiter Characters

The KeyQuoteChar, ValueQuoteChar, EscapeChar, KVDelimiter, and KVPDelimiter directives can be specified in several ways.

Unquoted single character

Any printable character can be specified as an unquoted character, except for the backslash (`\`):

```
Delimiter ;
```

Control characters

The following non-printable characters can be specified with escape sequences:

\a
: audible alert (bell)

\b
: backspace

\t
: horizontal tab

\n
: newline

\v
: vertical tab

\f
: formfeed

\r
: carriage return

For example, to use TAB delimiting:

```
Delimiter \t
```

A character in single quotes

The configuration parser strips whitespace, so it is not possible to define a space as the delimiter unless it is enclosed within quotes:

```
Delimiter ' '
```

Printable characters can also be enclosed:

```
Delimiter ';'
```

The backslash can be specified when enclosed within quotes:

```
Delimiter '\'
```

A character in double quotes

Double quotes can be used like single quotes:

```
Delimiter " "
```

The backslash can be specified when enclosed within double quotes:

```
Delimiter "\"
```

A hexadecimal ASCII code

Hexadecimal ASCII character codes can also be used by prepending `0x`. For example, the space can be specified as:

```
Delimiter 0x20
```

This is equivalent to:

```
Delimiter " "
```

4.7.2. Functions

The following functions are exported by *xm_kvp*.

string `to_kvp()`
: Convert the internal fields to a single key-value pair formatted string.

4.7.3. Procedures

The following procedures are exported by *xm_kvp*.

`parse_kvp();`
: Parse the `$raw_event` field as key-value pairs and populate the internal fields using the key names.

`parse_kvp(string source);`
: Parse the given string key-value pairs and populate the internal fields using the key names.

`reset_kvp();`
: Reset the KVP parser so that the autodetected KeyQuoteChar, ValueQuoteChar, KVDelimiter, and KVPDelimiter characters can be detected again.

`to_kvp();`
: Format the internal fields as key-value pairs and put this into the `$raw_event` field.

4.7.4. Examples

The following examples illustrate various scenarios for parsing KVPs, whether embedded, encapsulated (in Syslog, for example), or alone. In each case, the logs are converted from KVP input files to JSON output files, though obviously there are many other possibilities.

Example 58. Simple KVP Parsing

The following two lines of input are in a simple KVP format where each line consists of various keys with values assigned to them.

Input Sample

```
Name=John, Age=42, Weight=84, Height=142
Name=Mike, Weight=64, Age=24, Pet=dog, Height=172
```

This input can be parsed with the following configuration. The parsed fields can be used in NXLog expressions: a new field named `$Overweight` is added and set to TRUE if the conditions are met. Finally a few automatically added fields are removed, and the log is then converted to JSON.

nxlog.conf

```
<Extension kvp>
    Module          xm_kvp
    KVPDelimiter    ,
    KVDelimiter     =
    EscapeChar      \\
</Extension>

<Extension json>
    Module          xm_json
</Extension>

<Input filein>
    Module          im_file
    File            "modules/extension/kvp/xm_kvp5.in"
    <Exec>
        if $raw_event =~ /^#/ drop();
        else
        {
            kvp->parse_kvp();
            delete($EventReceivedTime);
            delete($SourceModuleName);
            delete($SourceModuleType);
            if ( integer($Weight) > integer($Height) - 100 ) $Overweight = TRUE;
            to_json();
        }
    </Exec>
</Input>

<Output fileout>
    Module          om_file
    File            'tmp/output'
</Output>

<Route parse_kvp>
    Path            filein => fileout
</Route>
```

Output Sample

```
{"Name":"John","Age":"42","Weight":"84","Height":"142","Overweight":true}
{"Name":"Mike","Weight":"64","Age":"24","Pet":"dog","Height":"172"}
```

Example 59. Parsing KVPs in Cisco ACS Syslog

The following lines are from a Cisco ACS source.

Input Sample

```
<38>2010-10-12 21:01:29 10.0.1.1 CisACS_02_FailedAuth 1k1fg93nk 1 0 Message-Type=Authen
failed,User-Name=John,NAS-IP-Address=10.0.1.2,AAA Server=acs01
<38>2010-10-12 21:01:31 10.0.1.1 CisACS_02_FailedAuth 2k1fg63nk 1 0 Message-Type=Authen
failed,User-Name=Foo,NAS-IP-Address=10.0.1.2,AAA Server=acs01
```

These logs are in Syslog format with a set of values present in each record and an additional set of KVPs. The following configuration can be used to process this and convert it to JSON.

nxlog.conf

```
<Extension json>
    Module          xm_json
</Extension>

<Extension syslog>
    Module          xm_syslog
</Extension>

<Extension kvp>
    Module          xm_kvp
    KVDelimiter     =
    KVPDelimiter    ,
</Extension>

<Input cisco>
    Module          im_file
    File            "modules/extension/kvp/cisco_acs.in"
    <Exec>
        parse_syslog_bsd();
        if ( $Message =~ /^CisACS_(\d\d)_(\S+) (\S+) (\d+) (\d+) (.*)$/ )
        {
            $ACSCategoryNumber = $1;
            $ACSCategoryName = $2;
            $ACSMessageId = $3;
            $ACSTotalSegments = $4;
            $ACSSegmentNumber = $5;
            $Message = $6;
            kvp->parse_kvp($Message);
        }
        else log_warning("does not match: " + to_json());
    </Exec>
</Input>

<Output file>
    Module          om_file
    File            "tmp/output"
    Exec            delete($EventReceivedTime);
    Exec            to_json();
</Output>

<Route cisco_to_file>
    Path            cisco => file
</Route>
```

Output Sample

```
{"SourceModuleName":"cisco","SourceModuleType":"im_file","SyslogFacilityValue":4,"SyslogFacilit
y":"AUTH","SyslogSeverityValue":6,"SyslogSeverity":"INFO","SeverityValue":2,"Severity":"INFO","
Hostname":"10.0.1.1","EventTime":"2010-10-12 21:01:29","Message":"Message-Type=Authen
failed,User-Name=John,NAS-IP-Address=10.0.1.2,AAA Server=acs01","ACSCategoryNumber":"02",
"ACSCategoryName":"FailedAuth","ACSMessageId":"1k1fg93nk","ACSTotalSegments":"1","ACSSegmentNum
ber":"0","Message-Type":"Authen failed","User-Name":"John","NAS-IP-Address":"10.0.1.2","AAA
Server":"acs01"}
{"SourceModuleName":"cisco","SourceModuleType":"im_file","SyslogFacilityValue":4,"SyslogFacilit
y":"AUTH","SyslogSeverityValue":6,"SyslogSeverity":"INFO","SeverityValue":2,"Severity":"INFO","
Hostname":"10.0.1.1","EventTime":"2010-10-12 21:01:31","Message":"Message-Type=Authen
failed,User-Name=Foo,NAS-IP-Address=10.0.1.2,AAA Server=acs01","ACSCategoryNumber":"02",
"ACSCategoryName":"FailedAuth","ACSMessageId":"2k1fg63nk","ACSTotalSegments":"1","ACSSegmentNum
ber":"0","Message-Type":"Authen failed","User-Name":"Foo","NAS-IP-Address":"10.0.1.2","AAA
Server":"acs01"}
```

Example 60. Parsing KVPs in Sidewinder Logs

The following line is from a Sidewinder log source.

Input Sample

```
date="May 5 14:34:40 2009
MDT",fac=f_mail_filter,area=a_kmvfilter,type=t_mimevirus_reject,pri=p_major,pid=10174,ruid=0,eu
id=0,pgid=10174,logid=0,cmd=kmvfilter,domain=MMF1,edomain=MMF1,message_id=(null),srcip=66.74.18
4.9,mail_sender=<habuzeid6@…>,virus_name=W32/Netsky.c@MM!zip,reason="Message scan detected a
Virus in msg Unknown, message being Discarded, and not quarantined"
```

This can be parsed and converted to JSON with the following configuration.

nxlog.conf

```
<Extension kvp>
    Module          xm_kvp
    KVPDelimiter    ,
    KVDelimiter     =
    EscapeChar      \\
    ValueQuoteChar  "
</Extension>

<Extension json>
    Module          xm_json
</Extension>

<Input sidewinder>
    Module          im_file
    File            "modules/extension/kvp/sidewinder.in"
    Exec            kvp->parse_kvp(); delete($EventReceivedTime); to_json();
</Input>

<Output file>
    Module          om_file
    File            'tmp/output'
</Output>

<Route sidewinder_to_file>
    Path            sidewinder => file
</Route>
```

Output Sample

```
{"SourceModuleName":"sidewinder","SourceModuleType":"im_file","date":"May 5 14:34:40 2009 MDT"
,"fac":"f_mail_filter","area":"a_kmvfilter","type":"t_mimevirus_reject","pri":"p_major","pid":"
10174","ruid":"0","euid":"0","pgid":"10174","logid":"0","cmd":"kmvfilter","domain":"MMF1","edom
ain":"MMF1","message_id":"(null)","srcip":"66.74.184.9","mail_sender":"<habuzeid6@…>","virus_na
me":"W32/Netsky.c@MM!zip","reason":"Message scan detected a Virus in msg Unknown, message being
Discarded, and not quarantined"}
```

Example 61. Parsing URL Request Parameters in Apache Access Logs

URLs in HTTP requests frequently contain URL parameters which are a special kind of key-value pairs delimited by the ampersand (&). Here is an example of two HTTP requests logged by the Apache web server in the Combined Log Format.

Input Sample

```
192.168.1.1 - foo [11/Jun/2013:15:44:34 +0200] "GET /do?action=view&obj_id=2 HTTP/1.1" 200 1514
"https://localhost" "Mozilla/5.0 (X11; Linux x86_64; rv:17.0) Gecko/17.0 Firefox/17.0"
192.168.1.1 - - [11/Jun/2013:15:44:44 +0200] "GET /do?action=delete&obj_id=42 HTTP/1.1" 401 788
"https://localhost" "Mozilla/5.0 (X11; Linux x86_64; rv:17.0) Gecko/17.0 Firefox/17.0"
```

The following configuration file parses the access log and extracts all the fields. The request parameters are extracted into the $HTTPParams field using a regular expression, and then this field is further parsed using the KVP parser. At the end of the processing all fields are converted to KVP format using the to_kvp() procedure of the *kvp2* instance.

nxlog.conf

```
<Extension kvp>
    Module          xm_kvp
    KVPDelimiter    &
    KVDelimiter     =
</Extension>

<Extension kvp2>
    Module          xm_kvp
    KVPDelimiter    ;
    KVDelimiter     =
    #QuoteMethod    None
</Extension>

<Input apache>
    Module          im_file
    File            "modules/extension/kvp/apache_url.in"
    <Exec>
        if $raw_event =~ /(?x)^(\S+)\ (\S+)\ (\S+)\ \[([^\]]+)\]\ \"(\S+)\ (.+)
                          \ HTTP.\d\.\d\"\ (\d+)\ (\d+)\ \"([^\"]+)\"\ \"([^\"]+)\"/
        {
            $Hostname = $1;
            if $3 != '-' $AccountName = $3;
            $EventTime = parsedate($4);
            $HTTPMethod = $5;
            $HTTPURL = $6;
            $HTTPResponseStatus = $7;
            $FileSize = $8;
            $HTTPReferer = $9;
            $HTTPUserAgent = $10;
            if $HTTPURL =~ /\?(.+)/ { $HTTPParams = $1; }
            kvp->parse_kvp($HTTPParams);
            delete($EventReceivedTime);
            kvp2->to_kvp();
        }
    </Exec>
</Input>

<Output file>
    Module          om_file
    File            'tmp/output'
</Output>

<Route apache_to_file>
    Path            apache => file
</Route>
```

The two request parameters *action* and *obj_id* then appear at the end of the KVP formatted lines.

Output Sample

```
SourceModuleName=apache;SourceModuleType=im_file;Hostname=192.168.1.1;AccountName=foo;EventTime
=2013-06-11
15:44:34;HTTPMethod=GET;HTTPURL=/do?action=view&obj_id=2;HTTPResponseStatus=200;FileSize=1514;H
TTPReferer=https://localhost;HTTPUserAgent='Mozilla/5.0 (X11; Linux x86_64; rv:17.0) Gecko/17.0
Firefox/17.0';HTTPParams=action=view&obj_id=2;action=view;obj_id=2;
SourceModuleName=apache;SourceModuleType=im_file;Hostname=192.168.1.1;EventTime=2013-06-11
15:44:44;HTTPMethod=GET;HTTPURL=/do?action=delete&obj_id=42;HTTPResponseStatus=401;FileSize=788
;HTTPReferer=https://localhost;HTTPUserAgent='Mozilla/5.0 (X11; Linux x86_64; rv:17.0)
Gecko/17.0 Firefox/17.0';HTTPParams=action=delete&obj_id=42;action=delete;obj_id=42;
```

NOTE	URL escaping is not handled.

4.8. Multi-Line Parser (xm_multiline)

This module can be used for parsing log messages that span multiple lines. All lines in an event are joined to form a single NXLog event record, which can be further processed as required. Each multi-line event is detected through some combination of header lines, footer lines, and fixed line counts, as configured. The name of the *xm_multiline* module instance is specified by the input module's InputType directive.

The module maintains a separate context for each input source, allowing multi-line messages to be processed correctly even when coming from multiple sources (specifically, multiple files or multiple network connections).

WARNING	UDP is treated as a single source and all logs are processed under the same context. It is therefore not recommended to use this module with im_udp if messages will be received by multiple UDP senders (such as Syslog).

4.8.1. Configuration

The *xm_multiline* module accepts the following directives in addition to the common module directives. One of FixedLineCount and HeaderLine must be specified.

FixedLineCount

This directive takes a positive integer number defining the number of lines to concatenate. This is useful when receiving log messages spanning a fixed number of lines. When this number is defined, the module knows where the event message ends and will not hold a message in the buffers until the next message arrives.

HeaderLine

This directive takes a string or a regular expression literal. This will be matched against each line. When the match is successful, the successive lines are appended until the next header line is read. This directive is mandatory unless FixedLineCount is used.

NOTE	Until a new message arrives with its associated header, the previous message is stored in the buffers because the module does not know where the message ends. The im_file module will forcibly flush this buffer after the configured PollInterval timeout. If this behavior is unacceptable, use an end marker with EndLine or switch to an encapsulation method (such as JSON).

EndLine

This is similar to the HeaderLine directive. This optional directive also takes a string or a regular expression literal to be matched against each line. When the match is successful the message is considered complete.

Exec

This directive is almost identical to the behavior of the Exec directive used by the other modules with the following differences:

- each line is passed in `$raw_event` as it is read, and the line terminator in included; and
- other fields cannot be used, and captured strings can not be stored as separate fields.

This is mostly useful for rewriting lines or filtering out certain lines with the drop() procedure.

4.8.2. Examples

Example 62. Parsing multi-line XML logs and converting to JSON

XML is commonly formatted as indented multi-line to make it more readable. In the following configuration file the HeaderLine and EndLine directives are used to parse the events. The events are then converted to JSON after some timestamp normalization.

nxlog.conf

```
<Extension multiline>
    Module          xm_multiline
    HeaderLine      /^<event>/
    EndLine         /^</event>/
</Extension>

<Extension xmlparser>
    Module          xm_xml
</Extension>

<Extension json>
    Module          xm_json
</Extension>

<Input filein>
    Module          im_file
    File            "modules/extension/multiline/xm_multiline5.in"
    InputType       multiline
    <Exec>
        # Discard everything that doesn't seem to be an xml event
        if $raw_event !~ /^<event>/ drop();

        # Parse the xml event
        parse_xml();

        # Rewrite some fields
        $EventTime = parsedate($timestamp);
        delete($timestamp);
        delete($EventReceivedTime);

        # Convert to JSON
        to_json();
    </Exec>
</Input>

<Output fileout>
    Module          om_file
    File            'tmp/output'
</Output>

<Route parse_xml>
    Path            filein => fileout
</Route>
```

Input Sample

```
<?xml version="1.0" encoding="UTF-8">
<event>
  <timestamp>2012-11-23 23:00:00</timestamp>
  <severity>ERROR</severity>
  <message>
    Something bad happened.
    Please check the system.
  </message>
</event>
<event>
  <timestamp>2012-11-23 23:00:12</timestamp>
  <severity>INFO</severity>
  <message>
   System state is now back to normal.
  </message>
</event>
```

Output Sample

```
{"SourceModuleName":"filein","SourceModuleType":"im_file","severity":"ERROR","message":"\n
Something bad happened.\n     Please check the system.\n   ","EventTime":"2012-11-23 23:00:00"}
{"SourceModuleName":"filein","SourceModuleType":"im_file","severity":"INFO","message":"\n
System state is now back to normal.\n  ","EventTime":"2012-11-23 23:00:12"}
```

Example 63. Parsing DICOM Logs

Each log message has a header (TIMESTAMP INTEGER SEVERITY) which is used as the message boundary. A regular expression is defined for this with the HeaderLine directive. Each log message is prepended with an additional line containing dashes and is written to a file.

nxlog.conf

```
<Extension dicom_multi>
    Module          xm_multiline
    HeaderLine      /^\d\d\d\d-\d\d-\d\d\d\d:\d\d:\d\d\.\d+\s+\d+\s+\S+\s+/
</Extension>

<Input filein>
    Module          im_file
    File            "modules/extension/multiline/xm_multiline4.in"
    InputType       dicom_multi
</Input>

<Output fileout>
    Module          om_file
    File            'tmp/output'
    Exec    $raw_event = "--------------------------------------\n" + $raw_event;
</Output>

<Route parse_dicom>
    Path            filein => fileout
</Route>
```

Input Sample

```
2011-12-1512:22:51.000000  4296   INFO   Association Request Parameters:
Our Implementation Class UID:    2.16.124.113543.6021.2
Our Implementation Version Name: RZDCX_2_0_1_8
Their Implementation Class UID:
Their Implementation Version Name:
Application Context Name:    1.2.840.10008.3.1.1.1
Requested Extended Negotiation: none
Accepted Extended Negotiation: none
2011-12-1512:22:51.000000  4296   DEBUG  Constructing Associate RQ PDU
2011-12-1512:22:51.000000  4296   DEBUG  WriteToConnection, length: 310, bytes written: 310,
loop no: 1
2011-12-1512:22:51.015000  4296   DEBUG  PDU Type: Associate Accept, PDU Length: 216 + 6 bytes
PDU header
  02  00  00  00  00  d8  00  01  00  00  50  41  43  53  20  20
  20  20  20  20  20  20  20  20  20  20  52  5a  44  43  58  20
  20  20  20  20  20  20  20  20  20  20  00  00  00  00  00  00
2011-12-1512:22:51.031000  4296   DEBUG  DIMSE sendDcmDataset: sending 146 bytes
```

Output Sample

```
--------------------------------------
2011-12-1512:22:51.000000  4296   INFO   Association Request Parameters:
Our Implementation Class UID:    2.16.124.113543.6021.2
Our Implementation Version Name: RZDCX_2_0_1_8
Their Implementation Class UID:
Their Implementation Version Name:
Application Context Name:    1.2.840.10008.3.1.1.1
Requested Extended Negotiation: none
Accepted Extended Negotiation: none
--------------------------------------
2011-12-1512:22:51.000000  4296   DEBUG  Constructing Associate RQ PDU
--------------------------------------
2011-12-1512:22:51.000000  4296   DEBUG  WriteToConnection, length: 310, bytes written: 310,
loop no: 1
--------------------------------------
2011-12-1512:22:51.015000  4296   DEBUG  PDU Type: Associate Accept, PDU Length: 216 + 6 bytes
PDU header
  02  00  00  00  00  d8  00  01  00  00  50  41  43  53  20  20
  20  20  20  20  20  20  20  20  20  20  52  5a  44  43  58  20
  20  20  20  20  20  20  20  20  20  20  00  00  00  00  00  00
--------------------------------------
2011-12-1512:22:51.031000  4296   DEBUG  DIMSE sendDcmDataset: sending 146 bytes
```

Example 64. Multi-line messages with a fixed string header

The following configuration will process messages having a fixed string header containing dashes. Each event is then prepended with a hash mark (#) and written to a file.

nxlog.conf

```
<Extension multiline>
    Module          xm_multiline
    HeaderLine      "---------------"
</Extension>

<Input filein>
    Module          im_file
    File            "modules/extension/multiline/xm_multiline1.in"
    InputType       multiline
    Exec            $raw_event = "#" + $raw_event;
</Input>

<Output fileout>
    Module          om_file
    File            'tmp/output'
</Output>

<Route parse_multiline>
    Path            filein => fileout
</Route>
```

Input Sample

```
---------------
1
---------------
1
2
---------------
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
ccccccccccccccccccccccccccccccccccccc
dddd
---------------
```

Output Sample

```
#---------------
1
#---------------
1
2
#---------------
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
ccccccccccccccccccccccccccccccccccccc
dddd
#---------------
```

Example 65. Multi-line messages with fixed line count

The following configuration will process messages having a fixed line count of four. Lines containing only whitespace are ignored and removed. Each event is then prepended with a hash mark (#) and written to a file.

nxlog.conf

```
<Extension multiline>
    Module          xm_multiline
    FixedLineCount  4
    Exec            if $raw_event =~ /^\s*$/ drop();
</Extension>

<Input filein>
    Module          im_file
    File            "modules/extension/multiline/xm_multiline2.in"
    InputType       multiline
</Input>

<Output fileout>
    Module          om_file
    File            'tmp/output'
    Exec            $raw_event = "#" + $raw_event;
</Output>

<Route parse_multiline>
    Path            filein => fileout
</Route>
```

Input Sample

```
1
2
3
4
1asd

2asdassad
3ewrwerew
4xcbccvbc

1dsfsdfsd
2sfsdfsdrewrwe

3sdfsdfsew
4werwerwrwe
```

Output Sample

```
#1
2
3
4
#1asd
2asdassad
3ewrwerew
4xcbccvbc
#1dsfsdfsd
2sfsdfsdrewrwe
3sdfsdfsew
4werwerwrwe
```

Example 66. Multi-line messages with a Syslog header

Often, multi-line messages are logged over Syslog and each line is processed as an event, with its own Syslog header. It is commonly necessary to merge these back into a single event message.

Input Sample

```
Nov 21 11:40:27 hostname app[26459]: Iface   MTU Met   RX-OK RX-ERR RX-DRP RX-OVR    TX-OK TX-
ERR TX-DRP TX-OVR Flg
Nov 21 11:40:27 hostname app[26459]: eth2       1500 0  16936814      0      0 0      30486067
0      8      0 BMRU
Nov 21 11:40:27 hostname app[26459]: lo        16436 0  277217234      0      0 0
277217234      0      0      0 LRU
Nov 21 11:40:27 hostname app[26459]: tun0       1500 0    316943      0      0 0        368642
0      0      0 MOPRU
Nov 21 11:40:28 hostname app[26459]: Iface   MTU Met   RX-OK RX-ERR RX-DRP RX-OVR    TX-OK TX-
ERR TX-DRP TX-OVR Flg
Nov 21 11:40:28 hostname app[26459]: eth2       1500 0  16945117      0      0 0      30493583
0      8      0 BMRU
Nov 21 11:40:28 hostname app[26459]: lo        16436 0  277217234      0      0 0
277217234      0      0      0 LRU
Nov 21 11:40:28 hostname app[26459]: tun0       1500 0    316943      0      0 0        368642
0      0      0 MOPRU
Nov 21 11:40:29 hostname app[26459]: Iface   MTU Met   RX-OK RX-ERR RX-DRP RX-OVR    TX-OK TX-
ERR TX-DRP TX-OVR Flg
Nov 21 11:40:29 hostname app[26459]: eth2       1500 0  16945270      0      0 0      30493735
0      8      0 BMRU
Nov 21 11:40:29 hostname app[26459]: lo        16436 0  277217234      0      0 0
277217234      0      0      0 LRU
Nov 21 11:40:29 hostname app[26459]: tun0       1500 0    316943      0      0 0        368642
0      0      0 MOPRU
```

The following configuration strips the Syslog header from the netstat output stored in the traditional Syslog formatted file, and each message is then printed again with a line of dashes used as a separator.

nxlog.conf

```
<Extension syslog>
    Module          xm_syslog
</Extension>

<Extension netstat>
    Module          xm_multiline
    FixedLineCount  4
    <Exec>
        parse_syslog_bsd();
        $raw_event = $Message + "\n";
    </Exec>
</Extension>

<Input filein>
    Module          im_file
    File            "modules/extension/multiline/xm_multiline3.in"
    InputType       netstat
</Input>

<Output fileout>
    Module          om_file
    File            'tmp/output'
    <Exec>
        $raw_event = "-------------------------------------------------------" +
                     "-----------------------------\n" + $raw_event;
    </Exec>
</Output>

<Route parse_multiline>
    Path            filein => fileout
</Route>
```

Output Sample

```
------------------------------------------------------------------------------------
Iface   MTU Met   RX-OK RX-ERR RX-DRP RX-OVR    TX-OK TX-ERR TX-DRP TX-OVR Flg
eth2       1500 0  16936814      0      0 0      30486067      0      8      0 BMRU
lo        16436 0  277217234      0      0 0      277217234      0      0      0 LRU
tun0       1500 0    316943      0      0 0        368642      0      0      0 MOPRU
------------------------------------------------------------------------------------
Iface   MTU Met   RX-OK RX-ERR RX-DRP RX-OVR    TX-OK TX-ERR TX-DRP TX-OVR Flg
eth2       1500 0  16945117      0      0 0      30493583      0      8      0 BMRU
lo        16436 0  277217234      0      0 0      277217234      0      0      0 LRU
tun0       1500 0    316943      0      0 0        368642      0      0      0 MOPRU
------------------------------------------------------------------------------------
Iface   MTU Met   RX-OK RX-ERR RX-DRP RX-OVR    TX-OK TX-ERR TX-DRP TX-OVR Flg
eth2       1500 0  16945270      0      0 0      30493735      0      8      0 BMRU
lo        16436 0  277217234      0      0 0      277217234      0      0      0 LRU
tun0       1500 0    316943      0      0 0        368642      0      0      0 MOPRU
```

4.9. Perl (xm_perl)

The Perl programming language is widely used for log processing and comes with a broad set of modules bundled or available from CPAN. Code can be written more quickly in Perl than in C, and code execution is safer because exceptions (croak/die) are handled properly and will only result in an unfinished attempt at log processing rather than taking down the whole NXLog process.

While the NXLog language is already a powerful framework, it is not intended to be a fully featured programming language and does not provide lists, arrays, hashes, and other features available in many high-level languages. With this module, Perl can be used to process event data via a built-in Perl interpreter.

The Perl interpreter is only loaded if the module is declared in the configuration. The module will parse the file specified in the PerlCode directive when NXLog starts the module. This file should contain one or more methods which can be called from the Exec directive of any module that will use Perl for log processing. See the example below.

To access event data, the Log::Nxlog module must be included, which provides the following methods.

log_debug(msg)
: Send the message *msg* to the internal logger on DEBUG log level. This method does the same as the log_debug() procedure in NXLog.

log_info(msg)
: Send the message *msg* to the internal logger on INFO log level. This method does the same as the log_info() procedure in NXLog.

log_warning(msg)
: Send the message *msg* to the internal logger on WARNING log level. This method does the same as the log_warning() procedure in NXLog.

log_error(msg)
: Send the message *msg* to the internal logger on ERROR log level. This method does the same as the log_error() procedure in NXLog.

delete_field(event, key)
: Delete the value associated with the field named *key*.

field_names(event)
: Return a list of the field names contained in the event data. This method can be used to iterate over all of the fields.

field_type(event, key)
: Return a string representing the type of the value associated with the field named *key*.

get_field(event, key)
: Retrieve the value associated with the field named *key*. This method returns a scalar value if the key exists and the value is defined, otherwise it returns undef.

set_field_boolean(event, key, value)
: Set the boolean value in the field named *key*.

set_field_integer(event, key, value)
: Set the integer value in the field named *key*.

set_field_string(event, key, value)
: Set the string value in the field named *key*.

For the full NXLog Perl API, see the POD documentation in `Nxlog.pm`. The documentation can be read with `perldoc Log::Nxlog`.

4.9.1. Configuration

The *xm_perl* module accepts the following directives in addition to the common module directives.

PerlCode

This mandatory directive expects a file containing valid Perl code. This file is read and parsed by the Perl interpreter. Methods defined in this file can be called with the call() procedure.

4.9.2. Procedures

The following procedures are exported by *xm_perl*.

`call(string subroutine);`

Call the given Perl *subroutine*.

`perl_call(string subroutine);`

Call the given Perl *subroutine*.

4.9.3. Examples

Example 67. Using the built-in Perl interpreter

In this example, logs are parsed as Syslog and then are passed to a Perl method which does a GeoIP lookup on the source address of the incoming message.

nxlog.conf

```
<Extension syslog>
    Module      xm_syslog
</Extension>

<Extension perl>
    Module      xm_perl
    PerlCode    modules/extension/perl/processlogs.pl
</Extension>

<Output fileout>
    Module      om_file
    File        'tmp/output'

    # First we parse the input natively from nxlog
    Exec        parse_syslog_bsd();

    # Now call the 'process' subroutine defined in 'processlogs.pl'
    Exec        perl_call("process");

    # You can also invoke this public procedure 'call' in case
    # of multiple xm_perl instances like this:
    # Exec      perl->call("process");
</Output>

<Route r>
    Path    filein => fileout
</Route>
```

processlogs.pl

```
use strict;
use warnings;

# Without Log::Nxlog you cannot access (read or modify) the event data
use Log::Nxlog;

use Geo::IP;

my $geoip;

BEGIN
{
    # This will be called once when nxlog starts so you can use this to
    # initialize stuff here
    #$geoip = Geo::IP->new(GEOIP_MEMORY_CACHE);
    $geoip = Geo::IP->open('modules/extension/perl/GeoIP.dat', GEOIP_MEMORY_CACHE);
}

# This is the method which is invoked from 'Exec' for each event
sub process
{
    # The event data is passed here when this method is invoked by the module
    my ( $event ) = @_;

    # We look up the county of the sender of the message
    my $msgsrcaddr = Log::Nxlog::get_field($event, 'MessageSourceAddress');
    if ( defined($msgsrcaddr) )
    {
    my $country = $geoip->country_code_by_addr($msgsrcaddr);
    $country = "unknown" unless ( defined($country) );
    Log::Nxlog::set_field_string($event, 'MessageSourceCountry', $country);
    }

    # Iterate over the fields
    foreach my $fname ( @{Log::Nxlog::field_names($event)} )
    {
    # Delete all fields except these
    if ( ! (($fname eq 'raw_event') ||
        ($fname eq 'AccountName') ||
        ($fname eq 'MessageSourceCountry')) )
    {
        Log::Nxlog::delete_field($event, $fname);
    }
    }

    # Check a field and rename it if it matches
    my $accountname = Log::Nxlog::get_field($event, 'AccountName');
    if ( defined($accountname) && ($accountname eq 'John') )
    {
    Log::Nxlog::set_field_string($event, 'AccountName', 'johnny');
    Log::Nxlog::log_info('renamed john');
    }
}
```

4.10. Syslog (xm_syslog)

This module provides support for the legacy BSD Syslog protocol as defined in RFC 3164 and the current IETF standard defined by RFCs 5424-5426. This is achieved by exporting functions and procedures usable from the

NXLog language. The transport is handled by the respective input and output modules (such as im_udp), this module only provides a parser and helper functions to create Syslog messages and handle facility and severity values.

The older but still widespread BSD Syslog standard defines both the format and the transport protocol in RFC 3164. The transport protocol is UDP, but to provide reliability and security, this line-based format is also commonly transferred over TCP and SSL. There is a newer standard defined in RFC 5424, also known as the IETF Syslog format, which obsoletes the BSD Syslog format. This format overcomes most of the limitations of BSD Syslog and allows multi-line messages and proper timestamps. The transport method is defined in RFC 5426 for UDP and RFC 5425 for TLS/SSL.

Because the IETF Syslog format supports multi-line messages, RFC 5425 defines a special format to encapsulate these by prepending the payload size in ASCII to the IETF Syslog message. Messages transferred in UDP packets are self-contained and do not need this additional framing. The following input reader and output writer functions are provided by the *xm_syslog* module to support this TLS transport defined in RFC 5425. While RFC 5425 explicitly defines that the TLS network transport protocol is to be used, pure TCP may be used if security is not a requirement. Syslog messages can also be written to file with this framing format using these functions.

InputType Syslog_TLS

This input reader function parses the payload size and then reads the message according to this value. It is required to support Syslog TLS transport defined in RFC 5425.

OutputType Syslog_TLS

This output writer function prepends the payload size to the message. It is required to support Syslog TLS transport defined in RFC 5425.

NOTE	The *Syslog_TLS* InputType/OutputType can work with any input/output such as im_tcp or im_file and does not depend on SSL transport at all. The name *Syslog_TLS* was chosen to refer to the octet-framing method described in RFC 5425 used for TLS transport.

NOTE	The pm_transformer module can also parse and create BSD and IETF Syslog messages, but the functions and procedures provided by this module make it possible to solve more complex tasks which pm_transformer is not capable of on its own.

Structured data in IETF Syslog messages is parsed and put into NXLog fields. The SD-ID will be prepended to the field name with a dot unless it is `NXLOG@XXXX`. Consider the following Syslog message:

```
<30>1 2011-12-04T21:16:10.000000+02:00 host app procid msgid [exampleSDID@32473
eventSource="Application" eventID="1011"] Message part
```

After this IETF-formatted Syslog message is parsed with parse_syslog_ietf(), there will be two additional fields: `$exampleSDID.eventID` and `$exampleSDID.eventSource`. When SD-ID is `NXLOG`, the field name will be the same as the SD-PARAM name. The two additional fields extracted from the structured data part of the following IETF Syslog message are `$eventID` and `$eventSource`:

```
<30>1 2011-12-04T21:16:10.000000+02:00 host app procid msgid [NXLOG@32473 eventSource="Application"
eventID="1011"] Message part
```

All fields in the structured data part are parsed as strings.

4.10.1. Configuration

The *xm_syslog* module accepts the following directives in addition to the common module directives.

IETFTimestampInGMT

This optional boolean directive can be used to format the timestamps produced by to_syslog_ietf() in UTC/GMT instead of local time. The default is FALSE: local time is used with a timezone indicator.

SnareDelimiter

This optional directive takes a single character (see below) as argument. This character is used by the to_syslog_snare() procedure to separate fields. If this directive is not specified, the default escape character is the tab (`\t`). In latter versions of Snare 4 this has changed to the hash mark (`#`); this directive can be used to specify the alternative delimiter. Note that there is no delimiter after the last field.

SnareReplacement

This optional directive takes a single character (see below) as argument. This character is used by the to_syslog_snare() procedure to replace occurrences of the delimiter character inside the `$Message` field. If this directive is not specified, the default replacement character is the space.

4.10.1.1. Specifying Quote, Escape, and Delimiter Characters

The SnareDelimiter and SnareReplacement directives can be specified in several ways.

Unquoted single character

Any printable character can be specified as an unquoted character, except for the backslash (`\`):

```
Delimiter ;
```

Control characters

The following non-printable characters can be specified with escape sequences:

\a

audible alert (bell)

\b

backspace

\t

horizontal tab

\n

newline

\v

vertical tab

\f

formfeed

\r

carriage return

For example, to use TAB delimiting:

```
Delimiter \t
```

A character in single quotes

The configuration parser strips whitespace, so it is not possible to define a space as the delimiter unless it is enclosed within quotes:

```
Delimiter ' '
```

Printable characters can also be enclosed:

```
Delimiter ';'
```

The backslash can be specified when enclosed within quotes:

```
Delimiter '\'
```

A character in double quotes

Double quotes can be used like single quotes:

```
Delimiter " "
```

The backslash can be specified when enclosed within double quotes:

```
Delimiter "\"
```

A hexadecimal ASCII code

Hexadecimal ASCII character codes can also be used by prepending `0x`. For example, the space can be specified as:

```
Delimiter 0x20
```

This is equivalent to:

```
Delimiter " "
```

4.10.2. Functions

The following functions are exported by *xm_syslog*.

string `syslog_facility_string(integer arg)`
Convert a Syslog facility value to a string.

integer `syslog_facility_value(string arg)`
Convert a Syslog facility string to an integer.

string `syslog_severity_string(integer arg)`
Convert a Syslog severity value to a string.

integer `syslog_severity_value(string arg)`
Convert a Syslog severity string to an integer.

4.10.3. Procedures

The following procedures are exported by *xm_syslog*.

`parse_syslog();`
Parse the $raw_event field as either BSD Syslog (RFC 3164) or IETF Syslog (RFC 5424) format.

`parse_syslog(string source);`
Parse the given string as either BSD Syslog (RFC 3164) or IETF Syslog (RFC 5424) format.

`parse_syslog_bsd();`
Parse the $raw_event field as BSD Syslog (RFC 3164) format.

`parse_syslog_bsd(string source);`
Parse the given string as BSD Syslog (RFC 3164) format.

parse_syslog_ietf();

Parse the $raw_event field as IETF Syslog (RFC 5424) format.

parse_syslog_ietf(string source);

Parse the given string as IETF Syslog (RFC 5424) format.

to_syslog_bsd();

Create a BSD Syslog formatted log message in $raw_event from the fields of the event. The following fields are used to construct the $raw_event field: $EventTime; $Hostname; $SourceName; $ProcessID; $Message or $raw_event; $SyslogSeverity, $SyslogSeverityValue, $Severity, or $SeverityValue; and $SyslogFacility or $SyslogFacilityValue. If the fields are not present, a sensible default is used.

to_syslog_ietf();

Create an IETF Syslog (RFC 5424) formatted log message in $raw_event from the fields of the event. The following fields are used to construct the $raw_event field: $EventTime; $Hostname; $SourceName; $ProcessID; $Message or $raw_event; $SyslogSeverity, $SyslogSeverityValue, $Severity, or $SeverityValue; and $SyslogFacility or $SyslogFacilityValue. If the fields are not present, a sensible default is used.

to_syslog_snare();

Create a SNARE Syslog formatted log message in $raw_event. The following fields are used to construct the $raw_event field: $EventTime, $Hostname, $SeverityValue, `$FileName`, `$EventID`, $SourceName ,`$AccountName`, `$AccountType`, `$EventType`, `$Category` and $Message.

4.10.4. Fields

The following fields are used by *xm_syslog*.

In addition to the fields listed below, the parse_syslog() and parse_syslog_ietf() procedures will create fields from the Structured Data part of an IETF Syslog message. If the SD-ID in this case is not "NXLOG", these fields will be prefixed by the SD-ID (for example, `$mySDID.CustomField`).

`$raw_event` (type: string)

A Syslog formatted string, set after to_syslog_bsd() or to_syslog_ietf() is called.

`$EventTime` (type: datetime)

The timestamp found in the Syslog message, set after parse_syslog(), parse_syslog_bsd(), or parse_syslog_ietf() is called. If the year value is missing, it is set to the current year.

`$Hostname` (type: string)

The hostname part of the Syslog line, set after parse_syslog(), parse_syslog_bsd(), or parse_syslog_ietf() is called.

`$Message` (type: string)

The message part of the Syslog line, set after parse_syslog(), parse_syslog_bsd(), or parse_syslog_ietf() is called.

`$MessageID` (type: string)

The MSGID part of the syslog message, set after parse_syslog_ietf() is called.

`$ProcessID` (type: string)

The process ID in the Syslog line, set after parse_syslog(), parse_syslog_bsd(), or parse_syslog_ietf() is called.

`$Severity` (type: string)

The normalized severity name of the event. See $SeverityValue.

`$SeverityValue` *(type: integer)*

The normalized severity number of the event, mapped as follows.

Syslog Severity	Normalized Severity
0/emerg	5/critical
1/alert	5/critical
2/crit	5/critical
3/err	4/error
4/warning	3/warning
5/notice	2/info
6/info	2/info
7/debug	1/debug

`$SourceName` *(type: string)*

The application/program part of the Syslog line, set after parse_syslog(), parse_syslog_bsd(), or parse_syslog_ietf() is called.

`$SyslogFacility` *(type: string)*

The facility name of the Syslog line, set after parse_syslog(), parse_syslog_bsd(), or parse_syslog_ietf() is called. The default facility is `user`.

`$SyslogFacilityValue` *(type: integer)*

The facility code of the Syslog line, set after parse_syslog(), parse_syslog_bsd(), or parse_syslog_ietf() is called. The default facility is `1` (user).

`$SyslogSeverity` *(type: string)*

The severity name of the Syslog line, set after parse_syslog(), parse_syslog_bsd(), or parse_syslog_ietf() is called. The default severity is `notice`. See $SeverityValue.

`$SyslogSeverityValue` *(type: integer)*

The severity code of the Syslog line, set after parse_syslog(), parse_syslog_bsd(), or parse_syslog_ietf() is called. The default severity is `5` (notice). See $SeverityValue.

4.10.5. Examples

Example 68. Sending a File as BSD Syslog over UDP

In this example, logs are collected from files, converted to BSD Syslog format with the to_syslog_bsd() procedure, and sent over UDP with the om_udp module.

nxlog.conf

```
<Extension syslog>
    Module  xm_syslog
</Extension>

<Input file>
    Module  im_file

    # We monitor all files matching the wildcard.
    # Every line is read into the $raw_event field.
    File    "/var/log/app*.log"

    <Exec>
        # Set the $EventTime field usually found in the logs by
        # extracting it with a regexp. If this is not set, the current
        # system time will be used which might be a little off.
        if $raw_event =~ /(\d\d\d\d\-\d\d-\d\d \d\d:\d\d:\d\d)/
        {
            $EventTime = parsedate($1);
        }

        # Now set the severity to something custom. This defaults to
        # 'INFO' if unset.
        if $raw_event =~ /ERROR/ $Severity = 'ERROR';
        else $Severity = 'INFO';

        # The facility can be also set, otherwise the default value is
        # 'USER'.
        $SyslogFacility = 'AUDIT';

        # The SourceName field is called the TAG in RFC 3164
        # terminology and is usually the process name.
        $SourceName = 'my_application';

        # It is also possible to rewrite the Hostname if you do not
        # want to use the system hostname.
        $Hostname = 'myhost';

        # The Message field is used if present, otherwise the current
        # $raw_event is prepended with the Syslog headers. You can do
        # some modifications on the Message if required. Here we add
        # the full path of the source file to the end of message line.
        $Message = $raw_event + ' [' + file_name() + ']';

        # Now create our RFC 3164 compliant Syslog line using the
        # fields set above and/or use sensible defaults where
        # possible. The result will be in $raw_event.
        to_syslog_bsd();
    </Exec>
</Input>

<Output udp>
    # This module just sends the contents of the $raw_event field to
    # the destination defined here, one UDP packet per message.
    Module  om_udp
    Host    192.168.1.42
    Port    1514
</Output>

<Route file_to_udp>
    Path    file => udp
</Route>
```

Example 69. Collecting BSD Style Syslog Messages over UDP

To collect BSD Syslog messages over UDP, use the parse_syslog_bsd() procedure coupled with the im_udp module as in the following example.

nxlog.conf

```
<Extension syslog>
    Module  xm_syslog
</Extension>

<Input udp>
    Module  im_udp
    Host    0.0.0.0
    Port    514
    Exec    parse_syslog_bsd();
</Input>

<Output file>
    Module  om_file
    File    "/var/log/logmsg.txt"
</Output>

<Route syslog_to_file>
    Path    udp => file
</Route>
```

Example 70. Collecting IETF Style Syslog Messages over UDP

To collect IETF Syslog messages over UDP as defined by RFC 5424 and RFC 5426, use the parse_syslog_ietf() procedure coupled with the im_udp module as in the following example. Note that, as for BSD Syslog, the default port is 514 (as defined by RFC 5426).

nxlog.conf

```
<Extension syslog>
    Module  xm_syslog
</Extension>

<Input ietf>
    Module  im_udp
    Host    0.0.0.0
    Port    514
    Exec    parse_syslog_ietf();
</Input>

<Output file>
    Module  om_file
    File    "/var/log/logmsg.txt"
</Output>

<Route ietf_to_file>
    Path    ietf => file
</Route>
```

Example 71. Collecting Both IETF and BSD Syslog Messages over the Same UDP Port

To collect both IETF and BSD Syslog messages over UDP, use the parse_syslog() procedure coupled with the im_udp module as in the following example. This procedure is capable of detecting and parsing both Syslog formats. Since 514 is the default UDP port number for both BSD and IETF Syslog, this port can be useful to collect both formats simultaneously. To accept both formats on different ports, the appropriate parsers can be used as in the previous two examples.

nxlog.conf

```
<Extension syslog>
    Module  xm_syslog
</Extension>

<Input udp>
    Module  im_udp
    Host    0.0.0.0
    Port    514
    Exec    parse_syslog();
</Input>

<Output file>
    Module  om_file
    File    "/var/log/logmsg.txt"
</Output>

<Route syslog_to_file>
    Path    udp => file
</Route>
```

Example 72. Collecting IETF Syslog Messages over TLS/SSL

To collect IETF Syslog messages over TLS/SSL as defined by RFC 5424 and RFC 5425, use the parse_syslog_ietf() procedure coupled with the im_ssl module as in this example. Note that the default port is 6514 in this case (as defined by RFC 5425). The payload format parser is handled by the Syslog_TLS input reader.

nxlog.conf

```
<Extension syslog>
    Module      xm_syslog
</Extension>

<Input ssl>
    Module      im_ssl
    Host        localhost
    Port        6514
    CAFile      %CERTDIR%/ca.pem
    CertFile    %CERTDIR%/client-cert.pem
    CertKeyFile %CERTDIR%/client-key.pem
    KeyPass     secret
    InputType   Syslog_TLS
    Exec        parse_syslog_ietf();
</Input>

<Output file>
    Module      om_file
    File        "/var/log/logmsg.txt"
</Output>

<Route ssl_to_file>
    Path        ssl => file
</Route>
```

Example 73. Forwarding IETF Syslog over TCP

The following configuration uses the to_syslog_ietf() procedure to convert input to IETF Syslog and forward it over TCP.

nxlog.conf

```
<Extension syslog>
    Module      xm_syslog
</Extension>

<Input file>
    Module      im_file
    File        "/var/log/input.txt"
    Exec        $TestField = "test value"; $Message = $raw_event;
</Input>

<Output tcp>
    Module      om_tcp
    Host        127.0.0.1
    Port        1514
    Exec        to_syslog_ietf();
    OutputType  Syslog_TLS
</Output>

<Route file_to_syslog>
    Path        file => tcp
</Route>
```

Because of the Syslog_TLS framing, the raw data sent over TCP will look like the following.

Output Sample

```
130 <13>1 2012-01-01T16:15:52.873750Z  - - - [NXLOG@14506 EventReceivedTime="2012-01-01
17:15:52" TestField="test value"] test message
```

This example shows that all fields—except those which are filled by the Syslog parser—are added to the structured data part.

Example 74. Conditional Rewrite of the Syslog Facility—Version 1

If the message part of the Syslog event matches the regular expression, the `$SeverityValue` field will be set to the "error" Syslog severity integer value (which is provided by the syslog_severity_value() function).

nxlog.conf

```
<Extension syslog>
    Module  xm_syslog
</Extension>

<Input udp>
    Module  im_udp
    Port    514
    Host    0.0.0.0
    Exec    parse_syslog_bsd();
</Input>

<Output file>
    Module  om_file
    File    "/var/log/logmsg.txt"
    Exec    if $Message =~ /error/ $SeverityValue = syslog_severity_value("error");
    Exec    to_syslog_bsd();
</Output>

<Route syslog_to_file>
    Path    udp => file
</Route>
```

Example 75. Conditional Rewrite of the Syslog Facility—Version 2

The following example does almost the same thing as the previous example, except that the Syslog parsing and rewrite is moved to a processor module and the rewrite only occurs if the facility was modified. This can make processing faster on multi-core systems because the processor module runs in a separate thread. This method can also minimize UDP packet loss because the input module does not need to parse Syslog messages and therefore can process UDP packets faster.

nxlog.conf

```
<Extension syslog>
    Module  xm_syslog
</Extension>

<Input udp>
    Module  im_udp
    Host    0.0.0.0
    Port    514
</Input>

<Processor rewrite>
    Module  pm_null
    <Exec>
        parse_syslog_bsd();
        if $Message =~ /error/
        {
            $SeverityValue = syslog_severity_value("error");
            to_syslog_bsd();
        }
    </Exec>
</Processor>

<Output file>
    Module  om_file
    File    "/var/log/logmsg.txt"
</Output>

<Route syslog_to_file>
    Path    udp => rewrite => file
</Route>
```

4.11. WTMP (xm_wtmp)

This module provides a parser function to process binary wtmp files. The module registers a parser function using the name of the extension module instance. This parser can be used as a parameter for the InputType directive in input modules such as im_file.

4.11.1. Configuration

The *xm_wtmp* module accepts only the common module directives.

4.11.2. Examples

Example 76. WTMP to JSON Format Conversion

The following configuration accepts WTMP and converts it to JSON.

nxlog.conf

```
<Extension wtmp>
    Module      xm_wtmp
</Extension>

<Extension json>
    Module      xm_json
</Extension>

<Input in>
    Module      im_file
    File        '/var/log/wtmp'
    InputType   wtmp
    Exec        to_json();
</Input>

<Output out>
    Module      om_file
    File        '/var/log/wtmp.txt'
</Output>

<Route processwtmp>
    Path        in => out
</Route>
```

Output Sample

```
{
  "EventTime":"2013-10-01 09:39:59",
  "AccountName":"root",
  "Device":"pts/1",
  "LoginType":"login",
  "EventReceivedTime":"2013-10-10 15:40:20",
  "SourceModuleName":"input",
  "SourceModuleType":"im_file"
}
{
  "EventTime":"2013-10-01 23:23:38",
  "AccountName":"shutdown",
  "Device":"no device",
  "LoginType":"shutdown",
  "EventReceivedTime":"2013-10-11 10:58:00",
  "SourceModuleName":"input",
  "SourceModuleType":"im_file"
}
```

4.12. XML (xm_xml)

This module provides functions and procedures for working with data formatted as Extensible Markup Language (XML). It can convert log messages to XML format and can parse XML into fields.

4.12.1. Configuration

The *xm_xml* module accepts only the common module directives.

4.12.2. Functions

The following functions are exported by *xm_xml*.

string `to_xml()`

Convert the fields to XML and returns this as a string value. The `$raw_event` field and any field having a leading dot (`.`) or underscore (`_`) will be automatically excluded.

4.12.3. Procedures

The following procedures are exported by *xm_xml*.

`parse_xml();`

Parse the `$raw_event` field as XML input.

`parse_xml(string source);`

Parse the given string as XML format.

`to_xml();`

Convert the fields to XML and put this into the `$raw_event` field. The `$raw_event` field and any field having a leading dot (`.`) or underscore (`_`) will be automatically excluded.

4.12.4. Examples

Example 77. Syslog to XML Format Conversion

The following configuration accepts Syslog (both BSD and IETF) and converts it to XML.

nxlog.conf

```
<Extension syslog>
    Module  xm_syslog
</Extension>

<Extension xml>
    Module  xm_xml
</Extension>

<Input tcp>
    Module  im_tcp
    Port    1514
    Host    0.0.0.0
    Exec    parse_syslog(); to_xml();
</Input>

<Output file>
    Module  om_file
    File    "/var/log/log.xml"
</Output>

<Route tcp_to_file>
    Path    tcp => file
</Route>
```

Input Sample

```
<30>Sep 30 15:45:43 host44.localdomain.hu acpid: 1 client rule loaded
```

Output Sample

```
<Event>
  <MessageSourceAddress>127.0.0.1</MessageSourceAddress>
  <EventReceivedTime>2012-03-08 15:05:39</EventReceivedTime>
  <SyslogFacilityValue>3</SyslogFacilityValue>
  <SyslogFacility>DAEMON</SyslogFacility>
  <SyslogSeverityValue>6</SyslogSeverityValue>
  <SyslogSeverity>INFO</SyslogSeverity>
  <SeverityValue>2</SeverityValue>
  <Severity>INFO</Severity>
  <Hostname>host44.localdomain.hu</Hostname>
  <EventTime>2012-09-30 15:45:43</EventTime>
  <SourceName>acpid</SourceName>
  <Message>1 client rule loaded</Message>
</Event>
```

Example 78. Converting Windows EventLog to Syslog-Encapsulated XML

The following configuration reads the Windows EventLog and converts it to the BSD Syslog format where the message part contains the fields in XML.

nxlog.conf

```
<Extension syslog>
    Module  xm_syslog
</Extension>

<Extension xml>
    Module  xm_xml
</Extension>

<Input eventlog>
    Module  im_msvistalog
    Exec    $Message = to_xml(); to_syslog_bsd();
</Input>

<Output tcp>
    Module  om_tcp
    Host    192.168.1.1
    Port    1514
</Output>

<Route eventlog_to_tcp>
    Path    eventlog => tcp
</Route>
```

Output Sample

```
<14>Mar  8 15:12:12 WIN-OUNNPISDHIG Service_Control_Manager: <Event><EventTime>2012-03-08
15:12:12</EventTime><EventTimeWritten>2012-03-08 15:12:12</EventTimeWritten><Hostname>WIN-
OUNNPISDHIG</Hostname><EventType>INFO</EventType><SeverityValue>2</SeverityValue><Severity>INFO
</Severity><SourceName>Service Control
Manager</SourceName><FileName>System</FileName><EventID>7036</EventID><CategoryNumber>0</Catego
ryNumber><RecordNumber>6791</RecordNumber><Message>The nxlog service entered the running state.
</Message><EventReceivedTime>2012-03-08 15:12:14</EventReceivedTime></Event>
```

Chapter 5. Input Modules

Input modules are responsible for collecting event log data from various sources.

Each module provides a set of fields for each log message, these are documented in the corresponding sections below. The NXLog core will add to this set the fields listed in the following section.

5.1. Fields

The following fields are used by *core*.

`$raw_event` *(type: string)*
: The data received from stream modules (im_file, im_tcp, etc.).

`$EventReceivedTime` *(type: datetime)*
: The time when the event is received. The value is not modified if the field already exists.

`$SourceModuleName` *(type: string)*
: The name of the module instance, for input modules. The value is not modified if the field already exists.

`$SourceModuleType` *(type: string)*
: The type of module instance (such as `im_file`), for input modules. The value is not modified if the field already exists.

5.2. DBI (im_dbi)

The *im_dbi* module allows NXLog to pull log data from external databases. This module utilizes the libdbi database abstraction library, which supports various database engines such as MySQL, PostgreSQL, MSSQL, Sybase, Oracle, SQLite, and Firebird. A SELECT statement can be specified, which will be executed periodically to check for new records.

NOTE	The *im_dbi* and om_dbi modules support GNU/Linux only because of the libdbi library. The *im_odbc* and *om_odbc* modules provide native database access on Windows (NXLog EE only).

NOTE	libdbi needs drivers to access the database engines. These are in the libdbd-* packages on Debian and Ubuntu. CentOS 5.6 has a libdbi-drivers RPM package, but this package does not contain any driver binaries under /usr/lib64/dbd. The drivers for both MySQL and PostgreSQL are in libdbi-dbd-mysql. If these are not installed, NXLog will return a libdbi driver initialization error.

5.2.1. Configuration

The *im_dbi* module accepts the following directives in addition to the common module directives.

Driver
: This mandatory directive specifies the name of the libdbi driver which will be used to connect to the database. A DRIVER name must be provided here for which a loadable driver module exists under the name `libdbdDRIVER.so` (usually under `/usr/lib/dbd/`). The MySQL driver is in the `libdbdmysql.so` file.

SQL
: This directive should specify the SELECT statement to be executed every PollInterval seconds. The module automatically appends a `WHERE id > ? LIMIT 10` clause to the statement. The result set returned by the SELECT statement must contain an *id* column which is then stored and used for the next query.

Option

This directive can be used to specify additional driver options such as connection parameters. The manual of the libdbi driver should contain the options available for use here.

PollInterval

This directive specifies how frequently the module will check for new records, in seconds. If this directive is not specified, the default is 1 second. Fractional seconds may be specified (`PollInterval 0.5` will check twice every second).

SavePos

If this boolean directive is set to TRUE, the position will be saved when NXLog exits. The position will be read from the cache file upon startup. The default is TRUE: the position will be saved if this directive is not specified. Even if **SavePos** is enabled, it can be explicitly turned off with the global NoCache directive.

5.2.2. Examples

Example 79. Reading From a MySQL Database

This example uses libdbi and the MySQL driver to connect to the logdb database on the local host and execute the provided statement.

nxlog.conf

```
<Input dbi>
    Module  im_dbi
    Driver  mysql
    Option  host 127.0.0.1
    Option  username mysql
    Option  password mysql
    Option  dbname logdb
    SQL     SELECT id, facility, severity, hostname, \
                   timestamp, application, message \
            FROM log
</Input>

<Output file>
    Module  om_file
    File    "tmp/output"
</Output>

<Route dbi_to_file>
    Path    dbi => file
</Route>
```

5.3. External Programs (im_exec)

This module will execute a program or script on startup and read its standard output. It can be used to easily integrate with exotic log sources which can be read only with the help of an external script or program.

WARNING	If you are using a Perl script, consider turning on *Autoflush* with `$| = 1;`, otherwise *im_exec* might not receive data immediately due to Perl's internal buffering. See the Perl language reference for more information about `$|`.

5.3.1. Configuration

The *im_exec* module accepts the following directives in addition to the common module directives. The Command directive is required.

Command

This mandatory directive specifies the name of the program or script to be executed.

Arg

This is an optional parameter. **Arg** can be specified multiple times, once for each argument that needs to be passed to the Command. Note that specifying multiple arguments with one **Arg** directive, with arguments separated by spaces, will not work (the Command would receive it as one argument).

InputType

See the InputType description in the global module configuration section.

Restart

Restart the process if it exits. There is a one second delay before it is restarted to avoid a denial-of-service when a process is not behaving. Looping should be implemented in the script itself, this directive is only to provide some safety against malfunctioning scripts and programs. This boolean directive defaults to FALSE: the Command will not be restarted if it exits.

5.3.2. Examples

Example 80. Emulating im_file

This configuration uses the tail command to read from a file.

NOTE	The im_file module should be used to read log messages from files. This example only demonstrates the use of the *im_exec* module.

nxlog.conf

```
<Input messages>
    Module  im_exec
    Command /usr/bin/tail
    Arg     -f
    Arg     /var/log/messages
</Input>

<Output file>
    Module  om_file
    File    "tmp/output"
</Output>

<Route messages_to_file>
    Path    messages => file
</Route>
```

5.4. Files (im_file)

This module can be used to read log messages from files. The file position can be persistently saved across restarts in order to avoid reading from the beginning again when NXLog is restarted. External rotation tools are also supported. When the module is not able to read any more data from the file, it checks whether the opened file descriptor belongs to the same filename it opened originally. If the inodes differ, the module assumes the file was moved and reopens its input.

im_file uses a one second interval to monitor files for new messages. This method was implemented because polling a regular file is not supported on all platforms. If there is no more data to read, the module will sleep for 1 second.

By using wildcards, the module can read multiple files simultaneously and will open new files as they appear. It will also enter newly created directories if recursion is enabled.

NOTE	The module needs to scan the directory content for wildcarded file monitoring. This can present a significant load if there are many files (hundreds or thousands) in the monitored directory. For this reason it is highly recommended to rotate files out of the monitored directory either using the built-in log rotation capabilities of NXLog or with external tools.

5.4.1. Configuration

The *im_file* module accepts the following directives in addition to the common module directives. The File directive is required.

File

This mandatory directive specifies the name of the input file to open. It must be a string type expression. For relative filenames you should be aware that NXLog changes its working directory to "/" unless the global SpoolDir is set to something else. On Windows systems the directory separator is the backslash (`\`). For compatibility reasons the forward slash (`/`) character can be also used as the directory separator, but this only works for filenames not containing wildcards. If the filename is specified using wildcards, the backslash (`\`) should be used for the directory separator.

Wildcards are supported in filenames only, directory names in the path cannot be wildcarded. Wildcards are not regular expressions, but are patterns commonly used by Unix shells to expand filenames (also known as "globbing").

?

Matches a single character only.

***

Matches zero or more characters.

\*

Matches the asterisk (`*`) character.

\?

Matches the question mark (`?`) character.

[…]

Used to specify a single character. The class description is a list containing single characters and ranges of characters separated by the hyphen (`-`). If the first character of the class description is `^` or `!`, the sense of the description is reversed (any character *not* in the list is accepted). Any character can have a backslash (`\`) preceding it, which is ignored, allowing the characters `]` and `-` to be used in the character class, as well as `^` and `!` at the beginning.

NOTE	The backslash (`\`) is used to escape the wildcard characters. Unfortunately this is the same as the directory separator on Windows. Take this into account when specifying wildcarded filenames on this platform. Suppose that log files under the directory `C:\test` need to be monitored. Specifying the wildcard `C:\test\*.log` will not match because `\*` becomes a literal asterisk and the filename is treated as non-wildcarded. For this reason the directory separator needs to be escaped: `C:\test\\*.log` will match our files. `C:\\test\\*.log` will also work. When specifying the filename using double quotes, this would became `C:\\test\\\\*.log` because the backslash is also used as an escape character inside double quoted string literals. Filenames on Windows systems are treated case-insensitively, but case-sensitively on Unix/Linux.

ActiveFiles

This directive specifies the maximum number of files NXLog will actively monitor. If there are modifications to more files in parallel than the value of this directive, then modifications to files above this limit will only get noticed after the DirCheckInterval (all data should be collected eventually). Typically there are only a few log sources actively appending data to log files, and the rest of the files are dormant after being rotated, so the default value of 10 files should be sufficient in most cases. This directive is also only relevant in case of a wildcarded File path.

CloseWhenIdle

If set to TRUE, this boolean directive specifies that open input files should be closed as soon as possible after there is no more data to read. Some applications request an exclusive lock on the log file when written or rotated, and this directive can possibly help if the application tries again to acquire the lock. The default is FALSE.

DirCheckInterval

This directive specifies how frequently, in seconds, the module will check the monitored directory for modifications to files and new files in case of a wildcarded File path. The default is twice the value of the PollInterval directive (if PollInterval is not set, the default is 2 seconds). Fractional seconds may be specified. It is recommended to increase the default if there are many files which cannot be rotated out and the NXLog process is causing high CPU load.

PollInterval

This directive specifies how frequently the module will check for new files and new log entries, in seconds. If this directive is not specified, it defaults to 1 second. Fractional seconds may be specified (`PollInterval 0.5` will check twice every second).

ReadFromLast

This optional boolean directive instructs the module to only read logs which arrived after NXLog was started if the saved position could not be read (for example on first start). When SavePos is TRUE and a previously saved position value could be read, the module will resume reading from this saved position. If **ReadFromLast** is FALSE, the module will read all logs from the file. This can result in quite a lot of messages, and is usually not the expected behavior. If this directive is not specified, it defaults to TRUE.

Recursive

If set to TRUE, this boolean directive specifies that input files should be searched recursively under sub-directories. This option takes effect only if wildcards are used in the filename. For example, if the File directive is set to `/var/log/*.log`, then `/var/log/apache2/access.log` will also match. Because directory wildcards are not supported, this directive only makes it possible to read multiple files from different sub-directories with a single im_file module instance. The default is TRUE.

RenameCheck

If set to TRUE, this boolean directive specifies that input files should be monitored for possible file rotation via renaming in order to avoid re-reading the file contents. A file is considered to be rotated when NXLog detects a new file whose inode and size matches that of another watched file which has just been deleted. Note that this does not always work correctly and can yield false positives when a log file is deleted and another is added with the same size. The file system is likely to reuse to inode number of the deleted file and thus the module will falsely detect this as a rename/rotation. For this reason the default value of **RenameCheck** is FALSE: renamed files are considered to be new and the file contents will be re-read.

NOTE	It is recommended to use a naming scheme for rotated files so names of rotated files do not match the wildcard and are not monitored anymore after rotation, instead of trying to solve the renaming issue with this directive.

SavePos

If this boolean directive is set to TRUE, the file position will be saved when NXLog exits. The file position will be read from the cache file upon startup. The default is TRUE: the file position will be saved if this directive is not specified. Even if **SavePos** is enabled, it can be explicitly turned off with the global NoCache directive.

5.4.2. Functions

The following functions are exported by *im_file*.

string *`file_name()`*

Return the name of the currently open file which the log was read from.

5.4.3. Examples

Example 81. Forwarding Logs From a File to a Remote Host

This configuration will read from a file and forward messages via TCP. No additional processing is done.

nxlog.conf

```
<Input messages>
    Module  im_file
    File    "/var/log/messages"
</Input>

<Output tcp>
    Module  om_tcp
    Host    192.168.1.1
    Port    514
</Output>

<Route messages_to_tcp>
    Path    messages => tcp
</Route>
```

5.5. Internal (im_internal)

NXLog produces its own logs about its operations, including errors and debug messages. This module makes it possible to insert those internal log messages into a route. Internal messages can also be generated from the NXLog language using the log_info(), log_warning(), and log_error() procedures.

NOTE	Only messages with log level INFO and above are supported. Debug messages are ignored due to technical reasons. For debugging purposes the direct logging facility should be used: see the global LogFile and LogLevel directives.

WARNING	One must be careful about the use of the *im_internal* module because it is easy to cause message loops. For example, consider the situation when internal log messages are sent to a database. If the database is experiencing errors which result in internal error messages, then these are again routed to the database and this will trigger further error messages, resulting in a loop. In order to avoid a resource exhaustion, the *im_internal* module will drop its messages when the queue of the next module in the route is full. It is recommended to always put the *im_internal* module instance in a separate route.

NOTE	If internal messages are required in Syslog format, they must be explicitly converted with pm_transformer or the to_syslog_bsd() procedure of the xm_syslog module, because the $raw_event field is not generated in Syslog format.

5.5.1. Configuration

The *im_internal* module accepts only the common module directives.

5.5.2. Fields

The following fields are used by *im_internal*.

`$raw_event` *(type: string)*
: The string passed to the log_info() or other log_* procedure.

`$ErrorCode` *(type: integer)*
: The error number provided by the Apache portable runtime library, if an error is logged resulting from an operating system error.

`$EventTime` *(type: datetime)*
: The current time.

`$Hostname` *(type: string)*
: The hostname where the log was produced.

`$Message` *(type: string)*
: The same value as $raw_event.

`$ProcessID` *(type: integer)*
: The process ID of the NXLog process.

`$Severity` *(type: string)*
: The severity name of the event.

`$SeverityValue` *(type: integer)*
: Depending on the log level of the internal message, the value corresponding to "debug", "info", "warning", "error", or "critical".

`$SourceName` *(type: string)*
: Set to `nxlog`.

5.5.3. Examples

Example 82. Forwarding Internal Messages over Syslog UDP

This configuration collects NXLog internal messages, adds BSD Syslog headers, and forwards via UDP.

nxlog.conf

```
<Extension syslog>
    Module  xm_syslog
</Extension>

<Input internal>
    Module  im_internal
</Input>

<Output udp>
    Module  om_udp
    Host    192.168.1.1
    Port    514
    Exec    to_syslog_bsd();
</Output>

<Route internal_to_udp>
    Path    internal => udp
</Route>
```

5.6. Kernel (im_kernel)

This module can collect kernel log messages from the kernel log buffer. Currently this module works on Linux only, where the klogctl() system call is used for this purpose. In order to be able to read kernel logs, special privileges are required. For this, NXLog needs to be started as root. Using the User and Group global directives NXLog can then drop its root privileges while keeping the CAP_SYS_ADMIN capability in order to read the kernel log buffer.

NOTE Unfortunately it is not possible to read from the /proc/kmsg pseudo file for an unprivileged process even if the CAP_SYS_ADMIN capability is kept. For this reason the /proc/kmsg interface is not supported by the *im_kernel* module. The im_file module should work fine with the /proc/kmsg pseudo file if one wishes to collect kernel logs this way, though this will require NXLog to be running as root.

Log Sample

```
<6>Some message from the kernel.
```

Kernel messages are valid BSD Syslog messages, with a priority from 0 (emerg) to 7 (debug), but do not contain timestamp and hostname fields. These can be parsed with the *xm_syslog* parse_syslog_bsd() procedure, and the timestamp and hostname fields will be added by NXLog.

5.6.1. Configuration

The *im_kernel* module accepts only the common module directives.

5.6.2. Examples

Example 83. Storing Raw Kernel Logs into a File

This configuration collects log messages from the kernel and writes them to file.

nxlog.conf

```
# drop privileges after being started as root
User nxlog
Group nxlog

<Input kernel>
    Module  im_kernel
</Input>

<Output file>
    Module  om_file
    File    "tmp/output"
</Output>

<Route kernel_to_file>
    Path    kernel => file
</Route>
```

5.7. Mark (im_mark)

Mark messages are used to indicate periodic activity to assure that the logger is running when there are no log messages coming in from other sources.

By default, if no module-specific directives are set, a log message will be generated every 30 minutes containing `-- MARK --`.

NOTE	The $raw_event field is not generated in Syslog format. If mark messages are required in Syslog format, they must be explicitly converted with the to_syslog_bsd() procedure.

NOTE	The functionality of the *im_mark* module can be also achieved using the Schedule block with a log_info("--MARK--") Exec statement, which would insert the messages via the im_internal module into a route. Using a single module for this task can simplify configuration.

5.7.1. Configuration

The *im_mark* module accepts the following directives in addition to the common module directives.

Mark

This optional directive sets the string for the mark message. The default is `-- MARK --`.

MarkInterval

This optional directive sets the interval for mark messages, in minutes. The default is 30 minutes.

5.7.2. Fields

The following fields are used by *im_mark*.

`$raw_event` *(type: string)*

The value defined by the Mark directive, `-- MARK --` by default.

`$EventTime` *(type: datetime)*
: The current time.

`$Message` *(type: string)*
: The same value as $raw_event.

`$ProcessID` *(type: integer)*
: The process ID of the NXLog process.

`$Severity` *(type: string)*
: The severity name: `INFO`.

`$SeverityValue` *(type: integer)*
: The INFO severity level value: `2`.

`$SourceName` *(type: string)*
: Set to `nxlog`.

5.7.3. Examples

Example 84. Using the im_mark Module

Here, NXLog will write the specified string to file every minute.

nxlog.conf

```
<Input mark>
    Module          im_mark
    MarkInterval    1
    Mark            -=| MARK |=-
</Input>

<Output file>
    Module          om_file
    File            "tmp/output"
</Output>

<Route mark_to_file>
    Path            mark => file
</Route>
```

5.8. Null (im_null)

This module does not generate any input, so basically it does nothing. Yet it can be useful for creating a dummy route, for testing purposes, or for Scheduled NXLog code execution. The *im_null* module accepts only the common module directives. See this example for usage.

5.9. TLS/SSL (im_ssl)

The *im_ssl* module uses the OpenSSL library to provide an SSL/TLS transport. It behaves like the im_tcp module, except that an SSL handshake is performed at connection time and the data is sent over a secure channel. Log messages transferred over plain TCP can be eavesdropped or even altered with a man-in-the-middle attack, while the *im_ssl* module provides a secure log message transport.

5.9.1. Configuration

The *im_ssl* module accepts the following directives in addition to the common module directives.

Host

The module will accept connections on this IP address or DNS hostname. The default is `localhost`.

Port

The module will listen for incoming connections on this port number. The default is port 514.

AllowUntrusted

This boolean directive specifies that the remote connection should be allowed without certificate verification. If set to TRUE the remote will be able to connect with an unknown or self-signed certificate. The default value is FALSE: all connections must present a trusted certificate.

CADir

This specifies the path to a directory containing certificate authority (CA) certificates, which will be used to check the certificate of the remote socket. The certificate filenames in this directory must be in the OpenSSL hashed format.

CAFile

This specifies the path of the certificate authority (CA) certificate, which will be used to check the certificate of the remote socket.

CertFile

This specifies the path of the certificate file to be used for the SSL handshake.

CertKeyFile

This specifies the path of the certificate key file to be used for the SSL handshake.

KeyPass

With this directive, a password can be supplied for the certificate key file defined in CertKeyFile. This directive is not needed for passwordless private keys.

CRLDir

This specifies the path to a directory containing certificate revocation lists (CRLs), which will be consulted when checking the certificate of the remote socket. The certificate filenames in this directory must be in the OpenSSL hashed format.

CRLFile

This specifies the path of the certificate revocation list (CRL) which will be consulted when checking the certificate of the remote socket.

RequireCert

This boolean value specifies that the remote must present a certificate. If set to TRUE and there is no certificate presented during the connection handshake, the connection will be refused. The default value is TRUE: each connection must use a certificate.

5.9.2. Fields

The following fields are used by *im_ssl*.

`$raw_event` (type: string)

The received string.

$MessageSourceAddress (type: *string*)

The IP address of the remote host.

5.9.3. Examples

Example 85. Accepting Binary Logs From Another NXLog Agent

This configuration accepts secured log messages in the NXLog binary format and writes them to file.

nxlog.conf

```
<Input ssl>
    Module      im_ssl
    Host        localhost
    Port        23456
    CAFile      %CERTDIR%/ca.pem
    CertFile    %CERTDIR%/client-cert.pem
    CertKeyFile %CERTDIR%/client-key.pem
    KeyPass     secret
    InputType   Binary
</Input>

<Output file>
    Module      om_file
    File        "tmp/output"
</Output>

<Route ssl_to_file>
    Path        ssl => file
</Route>
```

5.10. TCP (im_tcp)

This module accepts TCP connections on the configured address and port. It can handle multiple simultaneous connections. The TCP transfer protocol provides more reliable log transmission than UDP. If security is a concern, consider using the im_ssl module instead.

NOTE This module provides no access control. Firewall rules can be used to deny connections from certain hosts.

5.10.1. Configuration

The *im_tcp* module accepts the following directives in addition to the common module directives.

Host

The module will accept connections on this IP address or DNS hostname. For security, the default listen address is `localhost` (the *localhost* loopback address is not accessible from the outside). To receive logs from remote hosts, the address specified here must be accessible. The *any* address `0.0.0.0` is commonly used here.

Port

The module will listen for incoming connections on this port number. The default port is 514 if this directive is not specified.

5.10.2. Fields

The following fields are used by *im_tcp*.

`$raw_event` *(type: string)*
: The received string.

`$MessageSourceAddress` *(type: string)*
: The IP address of the remote host.

5.10.3. Examples

Example 86. Using the im_tcp Module

With this configuration, NXLog will listen for TCP connections on port 1514 and write received log messages to file.

nxlog.conf

```
<Input tcp>
    Module  im_tcp
    Host    0.0.0.0
    Port    1514
</Input>

<Output file>
    Module  om_file
    File    "tmp/output"
</Output>

<Route tcp_to_file>
    Path    tcp => file
</Route>
```

5.11. UDP (im_udp)

This module accepts UDP datagrams on the configured address and port. UDP is the transport protocol of the legacy BSD Syslog as described in RFC 3164, so this module can be particularly useful to receive such messages from older devices which do not support other transports.

WARNING	UDP is an unreliable transport protocol, and does not guarantee delivery. Messages may not be received or may be truncated. It is recommended to use the TCP or SSL transport modules instead, if possible.

To reduce the likelihood of message loss, consider:

- increasing the socket buffer size with SockBufSize,
- raising the route priority by setting the Priority directive (to a low number such as 1), and
- adding a pm_buffer instance.

NOTE	This module provides no access control. Firewall rules can be used to drop log events from certain hosts.

For parsing Syslog messages, see the pm_transformer module or the parse_syslog_bsd() procedure of xm_syslog.

5.11.1. Configuration

The *im_udp* module accepts the following directives in addition to the common module directives.

Host
: The module will accept messages on this IP address or DNS hostname. The default is `localhost`.

Port
: The module will listen for incoming connections on this port number. The default is port 514.

SockBufSize
: This optional directive sets the socket buffer size (SO_RCVBUF) to the value specified. If not set, the operating system defaults are used. If UDP packet loss is occurring at the kernel level, setting this to a high value (such as `150000000`) may help. On Windows systems the default socket buffer size is extremely low, and using this option is highly recommended.

5.11.2. Fields

The following fields are used by *im_udp*.

`$raw_event` *(type: string)*
: The received string.

`$MessageSourceAddress` *(type: string)*
: The IP address of the remote host.

5.11.3. Examples

Example 87. Using the im_udp Module

This configuration accepts log messages via UDP and writes them to file.

nxlog.conf

```
<Input udp>
    Module  im_udp
    Host    192.168.1.1
    Port    514
</Input>

<Output file>
    Module  om_file
    File    "tmp/output"
</Output>

<Route udp_to_file>
    Path    udp => file
</Route>
```

5.12. Unix Domain Sockets (im_uds)

This module allows log messages to be received over a Unix domain socket. Unix systems traditionally have a /dev/log or similar socket used by the system logger to accept messages. Applications use the syslog(3) system call to send messages to the system logger.

NOTE	This module supports SOCK_DGRAM type sockets only.

NOTE	It is recommended to disable FlowControl when this module is used to collect local Syslog messages from the /dev/log Unix domain socket. Otherwise, if the corresponding Output queue becomes full, the syslog() system call will block in any programs trying to write to the system log and an unresponsive system may result.

For parsing Syslog messages, see the pm_transformer module or the parse_syslog_bsd() procedure of xm_syslog.

5.12.1. Configuration

The *im_uds* module accepts the following directives in addition to the common module directives.

UDS

This specifies the path of the Unix domain socket. The default is `/dev/log`.

InputType

See the InputType directive in the list of common module directives. This defaults to `dgram`.

5.12.2. Examples

Example 88. Using the im_uds Module

This configuration will accept logs via the specified socket and write them to file.

nxlog.conf

```
<Input uds>
    Module      im_uds
    UDS         /dev/log
    FlowControl False
</Input>

<Output file>
    Module      om_file
    File        "/var/log/messages"
</Output>

<Route uds_to_file>
    Path        uds => file
</Route>
```

Chapter 6. Processor Modules

Processor modules can be used to process log messages in the log message path between configured Input and Output modules.

6.1. Blocker (pm_blocker)

This module blocks log messages and can be used to simulate a blocked route. When the module blocks the data flow, log messages are first accumulated in the buffers, and then the flow control mechanism pauses the input modules. Using the block() procedure, it is possible to programmatically stop or resume the data flow. It can be useful for real-world scenarios as well as testing. See the examples below. When the module starts, the blocking mode is disabled by default (it operates like pm_null would).

6.1.1. Configuration

The *pm_blocker* module accepts only the common module directives.

6.1.2. Functions

The following functions are exported by *pm_blocker*.

boolean `is_blocking()`
: Return TRUE if the module is currently blocking the data flow, FALSE otherwise.

6.1.3. Procedures

The following procedures are exported by *pm_blocker*.

`block(boolean mode);`
: When *mode* is TRUE, the module will block. A `block(FALSE)` should be called from a Schedule block or another module, it might not get invoked if the queue is already full.

6.1.4. Examples

Example 89. Using the pm_blocker Module

In this example messages are received over UDP and forwarded to another host via TCP. The log data is forwarded during non-working hours (between 7pm and 8am). During working hours, the data is buffered on the disk.

nxlog.conf

```
<Input udp>
    Module  im_udp
    Host    0.0.0.0
    Port    1514
</Input>

<Processor buffer>
    Module  pm_buffer
    # 100 MB disk buffer
    MaxSize 102400
    Type    disk
</Processor>

<Processor blocker>
    Module  pm_blocker
    <Schedule>
        When    0 8 * * *
        Exec    blocker->block(TRUE);
    </Schedule>
    <Schedule>
        When    0 19 * * *
        Exec    blocker->block(FALSE);
    </Schedule>
</Processor>

<Output tcp>
    Module  om_tcp
    Host    192.168.1.1
    Port    1514
</Output>

<Route udp_to_tcp>
    Path    udp => buffer => blocker => tcp
</Route>
```

6.2. Buffer (pm_buffer)

Messages received over UDP may be dropped by the operating system if packets are not read from the message buffer fast enough. Some logging subsystems using a small circular buffer can overwrite old logs in the buffer if it is not read, also resulting in loss of log data. Buffering can help in such situations.

The *pm_buffer* module supports disk- and memory-based log message buffering. If both are required, multiple *pm_buffer* instances can be used with different settings. Because a memory buffer can be faster, though its size is limited, combining memory and disk based buffering can be a good idea if buffering is frequently used.

The disk-based buffering mode stores the log message data in chunks. When all the data is successfully forwarded from a chunk, it is then deleted in order to save disk space.

NOTE	Using *pm_buffer* is only recommended when there is a chance of message loss. The built-in flow control in NXLog ensures that messages will not be read by the input module until the output side can send, store, or forward. When reading from files (with im_file) or the Windows EventLog (with im_mseventlog or im_msvistalog) it is rarely necessary to use the *pm_buffer* module unless log rotation is used. During a rotation, there is a possibility of dropping some data while the output module (im_tcp, for example) is being blocked.

6.2.1. Configuration

The *pm_buffer* module accepts the following directives in addition to the common module directives. The MaxSize and Type directives are required.

MaxSize

This mandatory directive specifies the size of the buffer in kilobytes.

Type

This directive can be set to either `Mem` or `Disk` to select memory- or disk-based buffering.

Directory

This directory will be used to store the disk buffer file chunks. This is only valid if Type is set to `Disk`.

WarnLimit

This directive specifies an optional limit, smaller than MaxSize, which will trigger a warning message when reached. The log message will not be generated again until the buffer size drops to half of **WarnLimit** and reaches it again in order to protect against a warning message flood.

6.2.2. Functions

The following functions are exported by *pm_buffer*.

integer `buffer_count()`

Return the number of log messages held in the memory buffer.

integer `buffer_size()`

Return the size of the memory buffer in bytes.

6.2.3. Examples

Example 90. Using a Memory Buffer to Protect Against UDP Message Loss

This configuration accepts log messages via UDP and forwards them via TCP. An intermediate memory-based buffer allows the im_udp module instance to continue accepting messages even if the om_tcp output stops working (caused by downtime of the remote host or network issues, for example).

nxlog.conf

```
<Input udp>
    Module      im_udp
    Host        0.0.0.0
    Port        514
</Input>

<Processor buffer>
    Module      pm_buffer
    # 1 MB buffer
    MaxSize     1024
    Type        Mem
    # warn at 512k
    WarnLimit   512
</Processor>

<Output tcp>
    Module      om_tcp
    Host        192.168.1.1
    Port        1514
</Output>

<Route udp_to_tcp>
    Path        udp => buffer => tcp
</Route>
```

6.3. Event Correlator (pm_evcorr)

The *pm_evcorr* module provides event correlation functionality in addition to the already available NXLog language features such as variables and statistical counters which can be also used for event correlation purposes.

This module was greatly inspired by the Perl based correlation tool SEC. Some of the rules of the *pm_evcorr* module were designed to mimic those available in SEC. This module aims to be a better alternative to SEC with the following advantages:

- The correlation rules in SEC work with the current time. With *pm_evcorr* it is possible to specify a time field which is used for elapsed time calculation making offline event correlation possible.
- SEC uses regular expressions extensively, which can become quite slow if there are many correlation rules. In contrast, this module can correlate pre-processed messages using fields from, for example, the pattern matcher and Syslog parsers without requiring the use of regular expressions (though these are also available for use by correlation rules). Thus testing conditions can be significantly faster when simple comparison is used instead of regular expression based pattern matching.
- This module was designed to operate on fields, making it possible to correlate structured logs in addition to simple free-form log messages.
- Most importantly, this module is written in C, providing performance benefits (where SEC is written in pure Perl).

The rulesets of this module can use a context. A context is an expression which is evaluated during runtime to a value and the correlation rule is checked in the context of this value. For example, to count the number of failed logins per user and alert if the failed logins exceed 3 for the user, the `$AccountName` would be used as the context. There is a separate context storage for each correlation rule instance. For global contexts accessible from all rule instances, see module variables and statistical counters.

6.3.1. Configuration

The *pm_evcorr* module accepts the following directives in addition to the common module directives.

The *pm_evcorr* configuration contains correlation rules which are evaluated for each log message processed by the module. Currently there are five rule types supported by pm_evcorr: Absence, Pair, Simple, Suppressed, and Thresholded. These rules are defined in configuration blocks. The rules are evaluated in the order they are defined. For example, a correlation rule can change a state, variable, or field which can be then used by a later rule. File inclusion can be useful to store correlation rules in a separate file.

Absence

This rule type does the opposite of Pair. When TriggerCondition evaluates to TRUE, this rule type will wait Interval seconds for RequiredCondition to become TRUE. If it does not become TRUE, it executes the statement(s) in the Exec directive(s).

Context

This optional directive specifies an expression to be used as the context. It must evaluate to a value. Usually a field is specified here.

Exec

One or more **Exec** directives must be specified, each taking a statement as argument.

NOTE	The evaluation of this Exec is not triggered by a log event; thus it does not make sense to use log data related operations such as accessing fields.

Interval

This mandatory directive takes an integer argument specifying the number of seconds to wait for RequiredCondition to become TRUE. Its value must be greater than 0. The TimeField directive is used to calculate time.

RequiredCondition

This mandatory directive takes an expression as argument which must evaluate to a boolean value. When this evaluates to TRUE after TriggerCondition evaluated to TRUE within Interval seconds, the statement(s) in the Exec directive(s) are NOT executed.

TriggerCondition

This mandatory directive takes an expression as argument which must evaluate to a boolean value.

Pair

When TriggerCondition evaluates to TRUE, this rule type will wait Interval seconds for RequiredCondition to become TRUE. It then executes the statement(s) in the Exec directive(s).

Context

This optional directive specifies an expression to be used as the context. It must evaluate to a value. Usually a field is specified here.

Exec

One or more **Exec** directives must be specified, each taking a statement as argument.

Interval

This directive takes an integer argument specifying the number of seconds to wait for RequiredCondition to become TRUE. If this directive is `0` or not specified, the rule will wait indefinitely for RequiredCondition to become TRUE. The TimeField directive is used to calculate time.

RequiredCondition

This mandatory directive takes an expression as argument which must evaluate to a boolean value. When this evaluates to TRUE after TriggerCondition evaluated to TRUE within Interval seconds, the statement(s) in the Exec directive(s) are executed.

TriggerCondition

This mandatory directive takes an expression as argument which must evaluate to a boolean value.

Simple

This rule type is essentially the same as the Exec directive supported by all modules. Because Execs are evaluated before the correlation rules, the **Simple** rule was also needed to be able to evaluate a statement as the other rules do, following the rule order. The **Simple** block has one directive also with the same name.

Exec

One or more **Exec** directives must be specified, with a statement as argument.

Stop

This rule will stop evaluating successive rules if the Condition evaluates to TRUE. The optional Exec directive will be evaluated in this case.

Condition

This mandatory directive takes an expression as argument which must evaluate to a boolean value. When it evaluates to TRUE, the correlation rule engine will stop checking any further rules.

Exec

One or more **Exec** directives may be specified, each taking a statement as argument. This will be evaluated when the specified Condition is satisfied. This directive is optional.

Suppressed

This rule type matches the given condition. If the condition evaluates to TRUE, the statement specified with the Exec directive is evaluated. The rule will then ignore any log messages for the time specified with Interval directive. This rule is useful for avoiding creating multiple alerts in a short period when a condition is satisfied.

Condition

This mandatory directive takes an expression as argument which must evaluate to a boolean value.

Context

This optional directive specifies an expression to be used as the context. It must evaluate to a value. Usually a field is specified here.

Exec

One or more **Exec** directives must be specified, each taking a statement as argument.

Interval

This mandatory directive takes an integer argument specifying the number of seconds to ignore the condition. The TimeField directive is used to calculate time.

Thresholded

This rule type will execute the statement(s) in the Exec directive(s) if the Condition evaluates to TRUE Threshold or more times during the Interval specified. The advantage of this rule over the use of statistical counters is that the time window is dynamic and shifts as log messages are processed.

Condition

This mandatory directive takes an expression as argument which must evaluate to a boolean value.

Context

This optional directive specifies an expression to be used as the context. It must evaluate to a value. Usually a field is specified here.

Exec

One or more **Exec** directives must be specified, each taking a statement as argument.

Interval

This mandatory directive takes an integer argument specifying a time window for Condition to become TRUE. Its value must be greater than 0. The TimeField directive is used to calculate time. This time window is dynamic, meaning that it will shift.

Threshold

This mandatory directive takes an integer argument specifying the number of times Condition must evaluate to TRUE within the given time Interval. When the threshold is reached, the module executes the statement(s) in the Exec directive(s).

ContextCleanTime

When a Context is used in the correlation rules, these must be purged from memory after they are expired, otherwise using too many context values could result in a high memory usage. This optional directive specifies the interval between context cleanups, in seconds. By default a `60` second cleanup interval is used if any rules use a Context and this directive is not specified.

TimeField

This specifies the name of the field to use for calculating elapsed time, such as `EventTime`. The name of the field must be specified without the leading dollar sign (`$`). If this parameter is not specified, the current time is assumed. This directive makes it possible to accurately correlate events based on the event time recorded in the logs and to do non-real-time event correlation.

6.3.2. Examples

Example 91. Correlation Rules

This following configuration sample contains a rule for each type.

nxlog.conf

```
<Input filein>
    Module          im_file
    File            "modules/processor/evcorr/testinput_evcorr2.txt"
    Exec    if ($raw_event =~ /^(\d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\d) (.+)/) {  \
                    $EventTime = parsedate($1);                               \
                    $Message = $2;                                            \
                    $raw_event = $Message;                                    \
                }
</Input>

<Input internal>
    Module          im_internal
    Exec            $raw_event = $Message;
    Exec            $EventTime = 2010-01-01 00:01:00;
</Input>

<Output fileout>
    Module          om_file
    File            'tmp/output'
</Output>

<Processor evcorr>
    Module          pm_evcorr
    TimeField       EventTime

    <Simple>
    Exec            if $Message =~ /^simple/ $raw_event = "got simple";
    </Simple>

    <Suppressed>
    # Match input event and execute an action list, but ignore the
    # following matching events for the next $Interval seconds.
    Condition       $Message =~ /^suppressed/
    Interval        30
    Exec            $raw_event = "suppressing..";
    </Suppressed>

    <Pair>
    # If TriggerCondition is true, wait Interval seconds for
    # RequiredCondition to be true and then do the Exec. If Interval is
    # 0, there is no window on matching.
    TriggerCondition    $Message =~ /^pair-first/
    RequiredCondition   $Message =~ /^pair-second/
    Interval            30
    Exec                $raw_event = "got pair";
    </Pair>

    <Absence>
    # If TriggerCondition is true, wait Interval seconds for
    # RequiredCondition to be true. If RequiredCondition does not become
    # true within the specified interval then do the Exec.
    TriggerCondition  $Message =~ /^absence-trigger/
    RequiredCondition $Message =~ /^absence-required/
    Interval    10
    Exec        log_info("'absence-required' not received within 10 secs");
    </Absence>

    <Thresholded>
    # If the number of events exceeds the given threshold within the
    # interval do the Exec. Same as SingleWithThreshold in SEC.
    Condition       $Message =~ /^thresholded/
    Threshold       3
    Interval        60
    Exec            $raw_event = "got thresholded";
    </Thresholded>

    <Stop>
    Condition       $EventTime < 2010-01-02 00:00:00
    Exec            log_debug("got stop");
    </Stop>

    <Simple>
    # This will be rewritten only if the previous Stop condition is
    # FALSE.
    Exec            $raw_event = "rewritten";
    </Simple>

</Processor>

<Route corr>
    Path            filein, internal => evcorr => fileout
</Route>
```

Input Sample

```
2010-01-01 00:00:00 Not simple
2010-01-01 00:00:01 suppressed1 - Suppress kicks in, will log 'suppressing..'
2010-01-01 00:00:10 simple1
2010-01-01 00:00:12 pair-first - now look for pair-second
2010-01-01 00:00:13 thresholded1
2010-01-01 00:00:15 thresholded2
2010-01-01 00:00:19 simple2
2010-01-01 00:00:20 thresholded3 - will log 'got thresholded'
2010-01-01 00:00:21 suppressed2 - suppressed and logged as is
2010-01-01 00:00:22 pair-second - will log 'got pair'
2010-01-01 00:00:23 suppressed3 - suppressed and logged as is
2010-01-01 00:00:25 pair-first
2010-01-01 00:00:26 absence-trigger
2010-01-01 00:00:29 absence-required - will not log 'got absence'
2010-01-01 00:00:46 absence-trigger
2010-01-01 00:00:56 pair-second - will not log 'got pair' because it is over the interval
2010-01-01 00:00:57 absence-required - will log an additional 'absence-required not received
within 10 secs'
2010-01-02 00:00:00 this will be rewritten
2010-01-02 00:00:10 this too
```

Output Sample

```
Not simple
suppressing..
got simple
pair-first - now look for pair-second
thresholded1
thresholded2
got simple
got thresholded
suppressed2 - suppressed and logged as is
got pair
suppressed3 - suppressed and logged as is
pair-first
absence-trigger
absence-required - will not log 'got absence'
absence-trigger
pair-second - will not log 'got pair' because it is over the interval
absence-required - will log an additional 'absence-required not received within 10 secs'
rewritten
rewritten
'absence-required' not received within 10 secs
```

6.4. Filter (pm_filter)

This is a simple module which forwards log messages if the specified condition is TRUE.

This module has been obsoleted by the NXLog language. Filtering is now possible in any module with a conditional drop() procedure in an Exec block or directive.

Example 92. Filtering Events With drop()

This statement drops the current event if the `$raw_event` field matches the specified regular expression.

```
if $raw_event =~ /^Debug/ drop();
```

6.4.1. Configuration

The *pm_filter* module accepts the following directives in addition to the common module directives.

Condition

This mandatory directive takes an expression as argument which must evaluate to a boolean value. If the expression does not evaluate to TRUE, the log message is discarded.

6.4.2. Examples

Example 93. Filtering Messages

This configuration retains only log messages that match one of the regular expressions, all others are discarded.

nxlog.conf

```
<Input uds>
    Module      im_uds
    UDS         /dev/log
</Input>

<Processor filter>
    Module      pm_filter
    Condition   $raw_event =~ /failed/ or $raw_event =~ /error/
</Processor>

<Output file>
    Module      om_file
    File        "/var/log/error"
</Output>

<Route uds_to_file>
    Path        uds => filter => file
</Route>
```

6.5. De-Duplicator (pm_norepeat)

This module can be used to filter out repeating messages. Like Syslog daemons, this module checks the previous message against the current. If they match, the current message is dropped. The module waits one second for duplicated messages to arrive. If duplicates are detected, the first message is forwarded, the rest are dropped, and a message containing "last message repeated n times" is sent instead.

6.5.1. Configuration

The *pm_norepeat* module accepts the following directives in addition to the common module directives.

CheckFields

This optional directive takes a comma-separated list of field names which are used to compare log messages. Only the fields listed here are compared, the others are ignored. For example, the `$EventTime` field will be different in repeating messages, so this field should not be used in the comparison. If this directive is not specified, the default field to be checked is `$Message`.

6.5.2. Fields

The following fields are used by *pm_norepeat*.

`$raw_event` *(type: string)*

A string containing the `last message repeated n times` message.

`$EventTime` *(type: datetime)*

The time of the last event or the current time if EventTime was not present in the last event.

`$Message` *(type: string)*

The same value as $raw_event.

$ProcessID *(type: integer)*
The process ID of the NXLog process.

$Severity *(type: string)*
The severity name: `INFO`.

$SeverityValue *(type: integer)*
The INFO severity level value: `2`.

$SourceName *(type: string)*
Set to `nxlog`.

6.5.3. Examples

Example 94. Filtering Out Duplicated Messages

This configuration reads log messages from the socket. The `$Hostname`, `$SourceName`, and `$Message` fields are used to detect duplicates. Then the messages are written to file.

nxlog.conf

```
<Input uds>
    Module      im_uds
    UDS         /dev/log
</Input>

<Processor norepeat>
    Module      pm_norepeat
    CheckFields Hostname, SourceName, Message
</Processor>

<Output file>
    Module      om_file
    File        "/var/log/messages"
</Output>


<Route uds_to_file>
    Path        uds => norepeat => file
</Route>
```

6.6. Null (pm_null)

This module does not do any special processing, so basically it does nothing. Yet it can be used with the Exec and Schedule directives, like any other module.

The *pm_null* module accepts only the common module directives.

See this example for usage.

6.7. Pattern Matcher (pm_pattern)

This module makes it possible to execute pattern matching with a pattern database file in XML format. Using this module is more efficient than having NXLog regular expression rules listed in Exec directives, because the *pm_pattern* module was designed in such a way that patterns do not need to be matched linearly. In addition, the module does an automatic on-the-fly pattern reordering internally for further speed improvements and has a feature which can be used to tag messages with additional fields useful for message classification.

There are other techniques such as the radix tree which solve the linearity problem; the drawback is that usually these require the user to learn a special syntax for specifying patterns. If the log message is already parsed and is not treated as single line of message, then it is possible to process only a subset of the patterns which partially solves the linearity problem. With other performance improvements employed within the *pm_pattern* module, its speed can compare to the other techniques. Yet the *pm_pattern* module uses regular expressions which are familiar to users and can easily be migrated from other tools.

Traditionally, pattern matching on log messages has employed a technique where the log message was one string and the pattern (regular expression or radix tree based pattern) was executed against it. To match patterns against logs which contain structured data (such as the Windows EventLog), this structured data (the fields of the log) must be converted to a single string. This is a simple but inefficient method used by many tools.

The NXLog patterns defined in the XML pattern database file can contain more than one field. This allows multi-dimensional pattern matching. Thus with NXLog's *pm_pattern* module there is no need to convert all fields into a single string as it can work with multiple fields.

Patterns can be grouped together under pattern groups. Pattern groups serve an optimization purpose. The group can have an optional *matchfield* block which can check a condition. If the condition (such as `$SourceName` matches `sshd`) is satisfied, the *pm_pattern* module will descend into the group and check each pattern against the log. If the pattern group's condition did not match (`$SourceName` was not `sshd`), the module can skip all patterns in the group without having to check each pattern individually.

When the *pm_pattern* module finds a matching pattern, the `$PatternID` and `$PatternName` fields are set on the log message. These can be used later in conditional processing and correlation rules of the pm_evcorr module, for example.

NOTE	The *pm_pattern* module does not process all patterns. It exits after the first matching pattern is found. This means that at most one pattern can match a log message. Multiple patterns that can match the same subset of logs should be avoided. For example, with two regular expression patterns `^\d+` and `^\d\d`, the second may never be matched because of the first. The internal order of patterns and pattern groups is changed dynamically by *pm_pattern*. Patterns with the highest match count are placed and tried first. In addition to performance optimization, setting the value of `$PatternID` would be problematic with multiple values because the language does not support arrays. For a strictly linearly executing pattern matcher, see the Exec directive.

6.7.1. Configuration

The *pm_pattern* module accepts the following directives in addition to the common module directives.

PatternFile
: This mandatory directive specifies the name of the pattern database file.

6.7.2. Fields

The following fields are used by *pm_pattern*.

`$PatternID` *(type: integer)*
: The ID number of the pattern which matched the message.

`$PatternName` *(type: string)*
: The name of the pattern which matched the message.

6.7.3. Examples

Example 95. Using the pm_pattern Module

This configuration reads BSD Syslog messages from the socket, processes the messages with a pattern file, and then writes them to file in JSON format.

nxlog.conf

```
<Extension json>
    Module      xm_json
</Extension>

<Extension syslog>
    Module      xm_syslog
</Extension>

<Input uds>
    Module      im_uds
    UDS         /dev/log
    Exec        parse_syslog_bsd();
</Input>

<Processor pattern>
    Module      pm_pattern
    PatternFile /var/lib/nxlog/patterndb.xml
</Processor>

<Output file>
    Module      om_file
    File        "/var/log/out"
    Exec        to_json();
</Output>

<Route uds_to_file>
    Path        uds => pattern => file
</Route>
```

The following pattern database contains two patterns to match SSH authentication messages. The patterns are under a group named *ssh* which checks whether the `$SourceName` field is `sshd` and only tries to match the patterns if the logs are indeed from sshd. The patterns both extract *AuthMethod*, *AccountName*, and *SourceIP4Address* from the log message when the pattern matches the log. Additionally *TaxonomyStatus* and *TaxonomyAction* are set. The second pattern utilizes the Exec block, which is evaluated when the pattern matches.

NOTE	For this pattern to work, the logs must be parsed with parse_syslog() prior to processing by the *pm_pattern* module (as in the above example), because it uses the `$SourceName` and `$Message` fields.

patterndb.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<patterndb>
 <created>2010-01-01 01:02:03</created>
 <version>42</version>

 <group>
   <name>ssh</name>
   <id>42</id>
   <matchfield>
    <name>SourceName</name>
    <type>exact</type>
    <value>sshd</value>
   </matchfield>

   <pattern>
    <id>1</id>
    <name>ssh auth success</name>

    <matchfield>
     <name>Message</name>
     <type>regexp</type>
        <!-- Accepted publickey for nxlogfan from 192.168.1.1 port 4242 ssh2 -->
     <value>^Accepted (\S+) for (\S+) from (\S+) port \d+ ssh2</value>
     <capturedfield>
    <name>AuthMethod</name>
    <type>string</type>
     </capturedfield>
     <capturedfield>
    <name>AccountName</name>
    <type>string</type>
     </capturedfield>
     <capturedfield>
    <name>SourceIP4Address</name>
        <type>string</type>
     </capturedfield>
    </matchfield>

    <set>
     <field>
       <name>TaxonomyStatus</name>
       <value>success</value>
       <type>string</type>
     </field>
     <field>
       <name>TaxonomyAction</name>
       <value>authenticate</value>
       <type>string</type>
     </field>
    </set>
   </pattern>

   <pattern>
    <id>2</id>
    <name>ssh auth failure</name>

    <matchfield>
     <name>Message</name>
     <type>regexp</type>
     <value>^Failed (\S+) for invalid user (\S+) from (\S+) port \d+ ssh2</value>

     <capturedfield>
    <name>AuthMethod</name>
    <type>string</type>
     </capturedfield>
     <capturedfield>
    <name>AccountName</name>
    <type>string</type>
     </capturedfield>
     <capturedfield>
    <name>SourceIP4Address</name>
        <type>string</type>
     </capturedfield>
    </matchfield>
```

```
      <set>
       <field>
         <name>TaxonomyStatus</name>
         <value>failure</value>
         <type>string</type>
       </field>
       <field>
         <name>TaxonomyAction</name>
         <value>authenticate</value>
         <type>string</type>
       </field>
      </set>

      <exec>
        $TestField = 'test';
      </exec>
      <exec>
        $TestField = $Testfield + 'value';
      </exec>
     </pattern>

   </group>

 </patterndb>
```

6.8. Format Converter (pm_transformer)

The *pm_transformer* module provides parsers for BSD Syslog, IETF Syslog, CSV, JSON, and XML formatted data and can also convert between. This module is now obsoleted by the functions and procedures provided by the following modules: xm_syslog, xm_csv, xm_json, and xm_xml. Using this module can be slightly faster than calling these procedures from an Exec directive.

6.8.1. Configuration

The *pm_transformer* module accepts the following directives in addition to the common module directives. For conversion to occur, the InputFormat and OutputFormat directives must be specified.

InputFormat

This directive specifies the input format of the `$raw_event` field so that it is further parsed into fields. If this directive is not specified, no parsing will be performed.

CSV

Input is parsed as a comma-separated list of values. See xm_csv for similar functionality. The input fields must be defined by CSVInputFields.

JSON

Input is parsed as JSON. This does the same as the parse_json() procedure.

syslog_bsd

Same as syslog_rfc3164.

syslog_ietf

Same as syslog_rfc5424.

syslog_rfc3164

Input is parsed in the BSD Syslog format as defined by RFC 3164. This does the same as the parse_syslog_bsd() procedure.

syslog_rfc5424

Input is parsed in the IETF Syslog format as defined by RFC 5424. This does the same as the parse_syslog_ietf() procedure.

XML

Input is parsed as XML. This does the same as the parse_xml() procedure.

OutputFormat

This directive specifies the output transformation. If this directive is not specified, fields are not converted and `$raw_event` is left unmodified.

CSV

Output in `$raw_event` is formatted as a comma-separated list of values. See xm_csv for similar functionality.

JSON

Output in `$raw_event` is formatted as JSON. This does the same as the to_json() procedure.

syslog_bsd

Same as syslog_rfc3164.

syslog_ietf

Same as syslog_rfc5424.

syslog_rfc3164

Output in `$raw_event` is formatted in the BSD Syslog format as defined by RFC 3164. This does the same as the to_syslog_bsd() procedure.

syslog_rfc5424

Output in `$raw_event` is formatted in the IETF Syslog format as defined by RFC 5424. This does the same as the to_syslog_ietf() procedure.

syslog_snare

Output in `$raw_event` is formatted in the SNARE Syslog format. This does the same as the to_syslog_snare() procedure. This should be used in conjunction with the im_mseventlog or im_msvistalog module to produce an output compatible with Snare Agent for Windows.

XML

Output in `$raw_event` is formatted in XML. This does the same as the to_xml() procedure.

CSVInputFields

This is a comma-separated list of fields which will be set from the input parsed. The field names must have the dollar sign (`$`) prepended.

CSVInputFieldTypes

This optional directive specifies the list of types corresponding to the field names defined in CSVInputFields. If specified, the number of types must match the number of field names specified with CSVInputFields. If this directive is omitted, all fields will be stored as strings. This directive has no effect on the fields-to-CSV conversion.

CSVOutputFields

This is a comma-separated list of message fields which are placed in the CSV lines. The field names must have the dollar sign ($) prepended.

6.8.2. Examples

Example 96. Using the pm_transformer Module

This configuration reads BSD Syslog messages from file and writes them to another file in CSV format.

nxlog.conf

```
<Extension syslog>
    Module          xm_syslog
</Extension>

<Input filein>
    Module          im_file
    File            "tmp/input"
</Input>

<Processor transformer>
    Module          pm_transformer
    InputFormat     syslog_rfc3164
    OutputFormat    csv
    CSVOutputFields $facility, $severity, $timestamp, $hostname, \
                    $application, $pid, $message
</Processor>

<Output fileout>
    Module          om_file
    File            "tmp/output"
</Output>

<Route filein_to_fileout>
    Path            filein => transformer => fileout
</Route>
```

Chapter 7. Output Modules

Output modules are responsible for writing event log data to various destinations.

7.1. Blocker (om_blocker)

This module is mostly for testing purposes. It will block log messages in order to simulate a blocked route, like when a network transport output module such as om_tcp blocks because of a network problem.

The sleep() procedure can also be used for testing by simulating log message delays.

7.1.1. Configuration

The *om_blocker* module accepts only the common module directives.

7.1.2. Examples

Example 97. Testing Buffering With the om_blocker Module

Because the route in this configuration is blocked, this will test the behavior of the configured memory-based buffer.

nxlog.conf

```
<Input uds>
    Module      im_uds
    UDS         /dev/log
</Input>

<Processor buffer>
    Module      pm_buffer
    WarnLimit   512
    MaxSize     1024
    Type        Mem
</Processor>

<Output blocker>
    Module      om_blocker
</Output>

<Route uds_to_blocker>
    Path        uds => buffer => blocker
</Route>
```

7.2. DBI (om_dbi)

The *om_dbi* module allows NXLog to store log data in external databases. This module utilizes the libdbi database abstraction library, which supports various database engines such as MySQL, PostgreSQL, MSSQL, Sybase, Oracle, SQLite, and Firebird. An INSERT statement can be specified, which will be executed for each log, to insert into any table schema.

NOTE The im_dbi and *om_dbi* modules support GNU/Linux only because of the libdbi library. The *im_odbc* and *om_odbc* modules provide native database access on Windows (available only in NXLog Enterprise Edition).

NOTE	libdbi needs drivers to access the database engines. These are in the libdbd-* packages on Debian and Ubuntu. CentOS 5.6 has a libdbi-drivers RPM package, but this package does not contain any driver binaries under /usr/lib64/dbd. The drivers for both MySQL and PostgreSQL are in libdbi-dbd-mysql. If these are not installed, NXLog will return a libdbi driver initialization error.

7.2.1. Configuration

The *om_dbi* module accepts the following directives in addition to the common module directives.

Driver

This mandatory directive specifies the name of the libdbi driver which will be used to connect to the database. A DRIVER name must be provided here for which a loadable driver module exists under the name `libdbdDRIVER.so` (usually under `/usr/lib/dbd/`). The MySQL driver is in the `libdbdmysql.so` file.

SQL

This directive should specify the INSERT statement to be executed for each log message. The field names (names beginning with `$`) will be replaced with the value they contain. String types will be quoted.

Option

This directive can be used to specify additional driver options such as connection parameters. The manual of the libdbi driver should contain the options available for use here.

7.2.2. Examples

These two examples are for the plain Syslog fields. Other fields generated by parsers, regular expression rules, the pm_pattern pattern matcher module, or input modules, can also be used. Notably, the im_msvistalog and im_mseventlog modules generate different fields than those shown in these examples.

Example 98. Storing Syslog in a PostgreSQL Database

Below is a table schema which can be used to store Syslog data:

```
CREATE TABLE log (
    id serial,
        timestamp timestamp  not null,
    hostname varchar(32) default NULL,
    facility varchar(10) default NULL,
    severity varchar(10) default NULL,
    application varchar(10) default NULL,
    message text,
    PRIMARY KEY (id)
);
```

The following configuration accepts log messages via TCP and uses libdbi to insert log messages into the database.

nxlog.conf

```
<Extension syslog>
    Module  xm_syslog
</Extension>

<Input tcp>
    Module  im_tcp
    Port    1234
    Host    0.0.0.0
    Exec    parse_syslog_bsd();
</Input>

<Output dbi>
    Module  om_dbi
    SQL     INSERT INTO log (facility, severity, hostname, timestamp, \
                             application, message) \
            VALUES ($SyslogFacility, $SyslogSeverity, $Hostname, '$EventTime', \
                    $SourceName, $Message)
    Driver  pgsql
    Option  host 127.0.0.1
    Option  username dbuser
    Option  password secret
    Option  dbname logdb
</Output>

<Route tcp_to_dbi>
    Path    tcp => dbi
</Route>
```

Example 99. Storing Logs in a MySQL Database

This configuration reads log messages from the socket and inserts them into a MySQL database.

nxlog.conf

```
<Extension syslog>
    Module      xm_syslog
</Extension>

<Input uds>
    Module      im_uds
    UDS         /dev/log
    Exec        parse_syslog_bsd();
</Input>

<Output dbi>
    Module      om_dbi
    SQL         INSERT INTO log (facility, severity, hostname, timestamp, \
                                 application, message) \
                VALUES ($SyslogFacility, $SyslogSeverity, $Hostname, '$EventTime', \
                        $SourceName, $Message)
    Driver      mysql
    Option      host 127.0.0.1
    Option      username mysql
    Option      password mysql
    Option      dbname logdb
</Output>

<Route uds_to_dbi>
    Path        uds => dbi
</Route>
```

7.3. Program (om_exec)

This module will execute a program or script on startup and write (pipe) log data to its standard input. Unless OutputType is set to something else, only the contents of the `$raw_event` field are sent over the pipe. The execution of the program or script will terminate when the module is stopped, which usually happens when NXLog exits and the pipe is closed.

NOTE The program or script is started when NXLog starts and must not exit until the module is stopped. To invoke a program or script for each log message, use xm_exec instead.

7.3.1. Configuration

The *om_exec* module accepts the following directives in addition to the common module directives. The Command directive is required.

Command

This mandatory directive specifies the name of the program or script to be executed.

Arg

This is an optional parameter. **Arg** can be specified multiple times, once for each argument that needs to be passed to the Command. Note that specifying multiple arguments with one **Arg** directive, with arguments separated by spaces, will not work (the Command will receive it as one argument).

7.3.2. Examples

Example 100. Piping Logs to an External Program

With this configuration, NXLog will start the specified command, read logs from socket, and write those logs to the standard input of the command.

nxlog.conf

```
<Input uds>
    Module  im_uds
    UDS     /dev/log
</Input>

<Output someprog>
    Module  om_exec
    Command /usr/bin/someprog
    Arg     -
</Output>

<Route uds_to_someprog>
    Path    uds => someprog
</Route>
```

7.4. Files (om_file)

This module can be used to write log messages to a file.

7.4.1. Configuration

The *om_file* module accepts the following directives in addition to the common module directives. The File directive is required.

File

This mandatory directive specifies the name of the output file to open. It must be a string type expression. If the expression in the **File** directive is not a constant string (it contains functions, field names, or operators), it will be evaluated before each event is written to the file (and after the Exec is evaluated). Note that the filename must be quoted to be a valid string literal, unlike in other directives which take a filename argument. For relative filenames, note that NXLog changes its working directory to "/" unless the global SpoolDir is set to something else.

Below are three variations for specifying the same output file on a Windows system:

```
File 'C:\logs\logmsg.txt'
File "C:\\logs\\logmsg.txt"
File 'C:/logs/logmsg.txt'
```

CreateDir

If set to TRUE, this optional boolean directive instructs the module to create the output directory before opening the file for writing if it does not exist. The default is FALSE.

OutputType

See the OutputType directive in the list of common module directives. If this directive is not specified the default is LineBased.

Sync

This optional boolean directive instructs the module to sync the file after each log message is written, ensuring that it is really written to disk from the buffers. Because this can hurt performance, the default is FALSE.

Truncate

This optional boolean directive instructs the module to truncate the file before each write, causing only the most recent log message to be saved. The default is FALSE: messages are appended to the output file.

7.4.2. Functions

The following functions are exported by *om_file*.

string `file_name()`

Return the name of the currently open file which was specified using the File directive. Note that this will be the old name if the filename changes dynamically; for the new name, use the expression specified for the File directive instead of using this function.

integer `file_size()`

Return the size of the currently open output file in bytes. Returns undef if the file is not open. This can happen if File is not a string literal expression and there was no log message.

7.4.3. Procedures

The following procedures are exported by *om_file*.

`reopen();`

Reopen the current file. This procedure should be called if the file has been removed or renamed, for example with the file_cycle(), file_remove(), or file_rename() procedures of the xm_fileop module. This does not need to be called after rotate_to() because that procedure reopens the file automatically.

`rotate_to(string filename);`

Rotate the current file to the *filename* specified. The module will then open the original file specified with the File directive. Note that the rename(2) system call is used internally which does not support moving files across different devices on some platforms. If this is a problem, first rotate the file on the same device. Then use the xm_exec exec_async() procedure to copy it to another device or file system, or use the xm_fileop file_copy() procedure.

7.4.4. Examples

Example 101. Storing Raw Syslog Messages into a File

This configuration reads log messages from socket and writes the messages to file. No additional processing is done.

nxlog.conf

```
<Input uds>
    Module  im_uds
    UDS     /dev/log
</Input>

<Output file>
    Module  om_file
    File    "/var/log/messages"
</Output>

<Route uds_to_file>
    Path    uds => file
</Route>
```

Example 102. File Rotation Based on Size

With this configuration, NXLog accepts log messages via TCP and parses them as BSD Syslog. A separate output file is used for log messages from each host. When the output file size exceeds 15 MB, it will be automatically rotated and compressed.

nxlog.conf

```
<Extension exec>
    Module  xm_exec
</Extension>

<Extension syslog>
    Module  xm_syslog
</Extension>

<Input tcp>
    Module  im_tcp
    Port    1514
    Host    0.0.0.0
    Exec    parse_syslog_bsd();
</Input>

<Output file>
    Module  om_file
    File    "tmp/output_" + $Hostname + "_" + month(now())
    <Exec>
        if file->file_size() > 15M
        {
            $newfile = "tmp/output_" + $Hostname + "_" +
                        strftime(now(), "%Y%m%d%H%M%S");
            file->rotate_to($newfile);
            exec_async("/bin/bzip2", $newfile);
        }
    </Exec>
</Output>

<Route tcp_to_file>
    Path    tcp => file
</Route>
```

7.5. HTTP(s) (om_http)

This module will connect to the specified URL in either plain HTTP or HTTPS mode. Each event is transferred in a single POST request. The module then waits for a response containing a successful status code (200, 201, or 202). It will reconnect and retry the delivery if the remote has closed the connection or a timeout is exceeded while waiting for the response. This HTTP-level acknowledgment ensures that no messages are lost during transfer.

7.5.1. Configuration

The *om_http* module accepts the following directives in addition to the common module directives. The URL directive is required.

URL

This mandatory directive specifies the URL where the module should POST the event data. The module operates in plain HTTP or HTTPS mode depending on the URL provided, and connects to the hostname specified in the URL. If the port number is not explicitly indicated in the URL, it defaults to port 80 for HTTP and port 443 for HTTPS.

ContentType

This directive sets the *Content-Type* HTTP header to the string specified. The *Content-Type* is set to `text/plain` by default.

HTTPSAllowUntrusted

This boolean directive specifies that the connection should be allowed without certificate verification. If set to TRUE, the connection will be allowed even if the remote HTTPS server presents an unknown or self-signed certificate. The default value is FALSE: the remote HTTPS server must present a trusted certificate.

HTTPSCADir

This specifies the path to a directory containing certificate authority (CA) certificates, which will be used to check the certificate of the remote HTTPS server. The certificate filenames in this directory must be in the OpenSSL hashed format.

HTTPSCAFile

This specifies the path of the certificate authority (CA) certificate, which will be used to check the certificate of the remote HTTPS server.

HTTPSCertFile

This specifies the path of the certificate file to be used for the HTTPS handshake.

HTTPSCertKeyFile

This specifies the path of the certificate key file to be used for the HTTPS handshake.

HTTPSCRLDir

This specifies the path to a directory containing certificate revocation lists (CRLs), which will be consulted when checking the certificate of the remote HTTPS server. The certificate filenames in this directory must be in the OpenSSL hashed format.

HTTPSCRLFile

This specifies the path of the certificate revocation list (CRL) which will be consulted when checking the certificate of the remote HTTPS server.

HTTPSKeyPass

With this directive, a password can be supplied for the certificate key file defined in HTTPSCertKeyFile. This directive is not needed for passwordless private keys.

7.5.2. Procedures

The following procedures are exported by *om_http*.

`set_http_request_path(string path);`

Set the *path* in the HTTP request to the string specified. This is useful if the URL is dynamic and parameters such as event ID need to be included in the URL. Note that the string must be URL encoded if it contains reserved characters.

7.5.3. Examples

Example 103. Sending Logs over HTTPS

This configuration reads log messages from file and forwards them via HTTPS.

nxlog.conf

```
<Input file>
    Module              im_file
    File                'input.log'
</Input>

<Output http>
    Module              om_http
    URL                 https://server:8080/
    HTTPSCertFile       %CERTDIR%/client-cert.pem
    HTTPSCertKeyFile    %CERTDIR%/client-key.pem
    HTTPSCAFile         %CERTDIR%/ca.pem
    HTTPSAllowUntrusted FALSE
</Output>

<Route file_to_http>
    Path                file => http
</Route>
```

7.6. Null (om_null)

Log messages sent to the *om_null* module instance are discarded, this module does not write its output anywhere. It can be useful for creating a dummy route, for testing purposes, or for Scheduled NXLog code execution. The *om_null* module accepts only the common module directives. See this example for usage.

7.7. TLS/SSL (om_ssl)

The *om_ssl* module uses the OpenSSL library to provide an SSL/TLS transport. It behaves like the om_tcp module, except that an SSL handshake is performed at connection time and the data is received over a secure channel. Log messages transferred over plain TCP can be eavesdropped or even altered with a man-in-the-middle attack, while the *om_ssl* module provides a secure log message transport.

7.7.1. Configuration

The *om_ssl* module accepts the following directives in addition to the common module directives. The Host directive is required.

Host
: The module will connect to this IP address or DNS hostname.

Port
: The module will connect to this port number on the remote host. The default is port 514.

AllowUntrusted
: This boolean directive specifies that the connection should be allowed without certificate verification. If set to TRUE the connection will be allowed even if the remote server presents an unknown or self-signed certificate. The default value is FALSE: the remote socket must present a trusted certificate.

CADir

This specifies the path to a directory containing certificate authority (CA) certificates, which will be used to check the certificate of the remote socket. The certificate filenames in this directory must be in the OpenSSL hashed format.

CAFile

This specifies the path of the certificate authority (CA) certificate, which will be used to check the certificate of the remote socket.

CertFile

This specifies the path of the certificate file to be used for the SSL handshake.

CertKeyFile

This specifies the path of the certificate key file to be used for the SSL handshake.

CRLDir

This specifies the path to a directory containing certificate revocation lists (CRLs), which will be consulted when checking the certificate of the remote socket. The certificate filenames in this directory must be in the OpenSSL hashed format.

CRLFile

This specifies the path of the certificate revocation list (CRL) which will be used to check the certificate of the remote socket against.

KeyPass

With this directive, a password can be supplied for the certificate key file defined in CertKeyFile. This directive is not needed for passwordless private keys.

OutputType

See the OutputType directive in the list of common module directives.

Reconnect

This directive has been deprecated as of version 2.4. The module will try to reconnect automatically at increasing intervals on all errors.

7.7.2. Procedures

The following procedures are exported by *om_ssl*.

`reconnect();`

Force a reconnection. This can be used from a Schedule block to periodically reconnect to the server.

7.7.3. Examples

Example 104. Sending Binary Data to Another NXLog Agent

This configuration reads log messages from socket and sends them in the NXLog binary format to another NXLog agent.

nxlog.conf

```
<Input uds>
    Module          im_uds
    UDS             tmp/socket
</Input>

<Output ssl>
    Module          om_ssl
    Host            localhost
    Port            23456
    CAFile          %CERTDIR%/ca.pem
    CertFile        %CERTDIR%/client-cert.pem
    CertKeyFile     %CERTDIR%/client-key.pem
    KeyPass         secret
    AllowUntrusted  TRUE
    OutputType      Binary
</Output>

<Route uds_to_ssl>
    Path            uds => ssl
</Route>
```

7.8. TCP (om_tcp)

This module initiates a TCP connection to a remote host and transfers log messages. Or, in Listen mode, this module accepts client connections and multiplexes data to all connected clients. The TCP transfer protocol provides more reliable log transmission than UDP. If security is a concern, consider using the om_ssl module instead.

7.8.1. Configuration

The *om_tcp* module accepts the following directives in addition to the common module directives. The Host directive is required.

Host

The module will connect to this IP address or DNS hostname. Or, if Listen is set to TRUE, the module will listen for connections on this address.

Port

The module will connect to this port number on the remote host. Or, if Listen is set to TRUE, the module will listen for connections on this port. The default is port 514.

Listen

If TRUE, this boolean directive specifies that *om_tcp* should listen for connections at the local address specified by the Host directive rather than opening a connection to the address. The default is FALSE: *om_tcp* will connect to the specified address.

OutputType

See the OutputType directive in the list of common module directives.

QueueInListenMode

If set to TRUE, this boolean directive specifies that events should be queued if no client is connected. If this module's buffer becomes full, the preceding module in the route will be paused or events will be dropped, depending on whether FlowControl is enabled. This directive only applies if Listen is set to TRUE. The default is FALSE: *om_tcp* will discard events if no client is connected.

Reconnect

This directive has been deprecated as of version 2.4. The module will try to reconnect automatically at increasing intervals on all errors.

7.8.2. Procedures

The following procedures are exported by *om_tcp*.

`reconnect();`

Force a reconnection. This can be used from a Schedule block to periodically reconnect to the server.

7.8.3. Examples

Example 105. Transferring Raw Logs over TCP

With this configuration, NXLog will read log messages from socket and forward them via TCP.

nxlog.conf

```
<Input uds>
    Module  im_uds
    UDS     /dev/log
</Input>

<Output tcp>
    Module  om_tcp
    Host    192.168.1.1
    Port    1514
</Output>

<Route uds_to_tcp>
    Path    uds => tcp
</Route>
```

7.9. UDP (om_udp)

This module sends log messages as UDP datagrams to the address and port specified. UDP is the transport protocol of the legacy BSD Syslog standard as described in RFC 3164, so this module can be particularly useful to send messages to devices or Syslog daemons which do not support other transports.

7.9.1. Configuration

The *om_udp* module accepts the following directives in addition to the common module directives. The Host directive is required.

Host

The module will connect to this IP address or DNS hostname.

Port

The module will connect to this port number on the remote host. The default is port 514.

SockBufSize

This optional directive sets the socket buffer size (SO_SNDBUF) to the value specified. If this is not set, the operating system default is used.

7.9.2. Examples

Example 106. Sending Raw Syslog over UDP

This configuration reads log messages from socket and forwards them via UDP.

nxlog.conf

```
<Input uds>
    Module  im_uds
    UDS     /dev/log
</Input>

<Output udp>
    Module  om_udp
    Host    192.168.1.1
    Port    1514
</Output>

<Route uds_to_udp>
    Path    uds => udp
</Route>
```

7.10. Unix Domain Sockets (om_uds)

This module allows log messages to be sent to a Unix domain socket. Unix systems traditionally have a /dev/log or similar socket used by the system logger to accept messages. Applications use the syslog(3) system call to send messages to the system logger. NXLog can use this module to send log messages to another Syslog daemon via the socket.

NOTE	This module supports SOCK_DGRAM type sockets only. SOCK_STREAM type sockets may be supported in the future.

7.10.1. Configuration

The *om_uds* module accepts the following directives in addition to the common module directives.

UDS

This specifies the path of the Unix domain socket. The default is `/dev/log`.

7.10.2. Examples

Example 107. Using the om_uds Module

This configuration reads log messages from a file, adds BSD Syslog headers with default fields, and writes the messages to socket.

nxlog.conf

```
<Extension syslog>
    Module  xm_syslog
</Extension>

<Input file>
    Module  im_file
    File    "/var/log/custom_app.log"
</Input>

<Output uds>
    Module  om_uds
    # Defaulting Syslog fields and creating Syslog output
    Exec    parse_syslog_bsd(); to_syslog_bsd();
    UDS     /dev/log
</Output>

<Route file_to_uds>
    Path    file => uds
</Route>
```