NAME

mib2c.conf - How to write mib2c.conf files to do ANYTHING based on MIB input.

SYNOPSIS

% cat > mib2c.test.conf << EOF @foreach $t table@
 Starting table $t
 @foreach $c column@
   echo $t has column $c which has a syntax of $c.syntax
 @end@

@end@ EOF

% mib2c −c mib2c.test.conf internet

DESCRIPTION

The mib2c.conf script language is a MIB-particular language designed to easily process MIB nodes in ways that you want. mib2c is a misnomer (for historical purposes), because you can produce anything (not just C code). Look in the Net-SNMP "local" directory for a bunch of example mib2c.*.conf files and behold the power before you.

COMMANDS

All commands within mib2c.conf files are embraced by @ signs. Anything with an @ sign at the front and back of the line is generally supposed to be a mib2c specific command. These are detailed here:

@open FILE@
>   writes generated output to FILE note that for file specifications, opening '-' will print to stdout.

@append FILE@
>   appends the given FILE

@close FILE@
>   closes the given FILE

@push@
>   save the current outputs, then clear outputs. Use with @open@ and @pop@ to write to a new file without interfering with current outputs.

@pop@
>   pop up the process() stack one level. Use after a @push@ to return to the previous set of open files.

@foreach $VAR scalar@
>   repeat iterate over code until @end@ setting $VAR to all known scalars

@foreach $VAR table@
>   repeat iterate over code until @end@ setting $VAR to all known tables

@foreach $VAR column@
>   repeat iterate over code until @end@ setting $VAR to all known columns within a given table. Obviously this must be called within a foreach-table clause.

@foreach $VAR nonindex@
>   repeat iterate over code until @end@ setting $VAR to all known non-index columns within a given table. Obviously this must be called within a foreach-table clause.

@foreach $VAR internalindex@
>   repeat iterate over code until @end@ setting $VAR to all known internal index columns within a given table. Obviously this must be called within a foreach-table clause.

@foreach $VAR externalindex@
>   repeat iterate over code until @end@ setting $VAR to all known external index columns within a given table. Obviously this must be called within a foreach-table clause.

@foreach $VAR index@
>   repeat iterate over code until @end@ setting $VAR to all known indexes within a given table. Obviously this must be called within a foreach-table clause.

@foreach $VAR notifications@
>   repeat iterate over code until @end@ setting $VAR to all known notifications

@foreach $VAR varbinds@
>   repeat iterate over code until @end@ setting $VAR to all known varbinds Obviously this must be called within a foreach-notifications clause.

@foreach $LABEL, $VALUE enum@
>   repeat iterate over code until @end@ setting $LABEL and $VALUE to the label and values from the enum list.

@foreach $RANGE_START, $RANGE_END range NODE@
>   repeat iterate over code until @end@ setting $RANGE_START and $RANGE_END to the legal accepted range set for a given mib NODE.

@foreach $var stuff a b c d@
>   repeat iterate over values a, b, c, d as assigned generically (ie, the values are taken straight from the list with no mib-expansion, etc).

@while expression@
>   repeat iterate over code until the expression is false

@eval $VAR = expression@
>   evaluates expression and assigns the results to $VAR. This is not a full perl eval, but sort of a ""psuedo"" eval useful for simple expressions while keeping the same variable name space. See below for a full-blown export to perl.

@perleval STUFF@
>   evaluates STUFF directly in perl. Note that all mib2c variables interpereted within .conf files are in $vars{NAME} and that a warning will be printed if STUFF does not return 0. (adding a 'return 0;' at the end of STUFF is a workaround.

@startperl@

@endperl@
>   treats everything between these tags as perl code, and evaluates it.

@next@
>   restart foreach; should only be used inside a conditional. skips out of current conditional, then continues to skip to end for the current foreach clause.

@if expression@
>   evaluates expression, and if expression is true processes contained part until appropriate @end@ is reached. If the expression is false, the next @elsif expression@ expression (if it exists) will be evaluated, until an expression is true. If no such expression exists and an @else@ clause is found, it will be evaluated.

@ifconf file@
>   If the specified file can be found in the conf file search path, and if found processes contained part until an appropriate @end@ is found. As with a regular @if expression@, @elsif expression@ and @else@ can be used.

@ifdir dir@
>   If the specified directory exists, process contained part until an appropriate @end@ is found. As with a regular @if expression@, @elsif expression@ and @else@ can be used.

@define NAME@

@enddefine@
>   Memorizes ""stuff"" between the define and enddefine tags for later calling as NAME by @calldefine NAME@.

@calldefine NAME@
>   Executes stuff previously memorized as NAME.

@printf "expression" stuff1, stuff2, ...@
>   Like all the other printf's you know and love.

@run FILE@
>   Sources the contents of FILE as a mib2c file, but does not affect current files opened.

@include FILE@
>   Sources the contents of FILE as a mib2c file and appends its output to the current output.

@prompt $var QUESTION@
>   Presents the user with QUESTION, expects a response and puts it in $var

@print STUFF@
>   Prints stuff directly to the users screen (ie, not to where normal mib2c output goes)

@quit@
>   Bail out (silently)

@exit@
>   Bail out!

VARIABLES

Variables in the mib2c language look very similar to perl variables, in that they start with a "$". They can be used for anything you want, but most typically they'll hold mib node names being processed by @foreach ...@ clauses.

They also have a special properties when they are a mib node, such that adding special suffixes to them will allow them to be interpreted in some fashion. The easiest way to understand this is through an example. If the variable 'x' contained the word 'ifType' then some magical things happen. In mib2c output, anytime $x is seen it is replaced with "ifType". Additional suffixes can be used to get other aspects of that mib node though. If $x.objectID is seen, it'll be replaced by the OID for ifType: ".1.3.6.1.2.1.2.2.1.3". Other suffixes that can appear after a dot are listed below.

One last thing: you can use things like $vartext immediately ending in some other text, you can use { }s to get proper expansion of only part of the mib2c input. IE, $xtext will produce "$xtext", but ${x}text will produce "ifTypetext" instead.

$var.uc   all upper case version of $var

$var.objectID
>   dotted, fully-qualified, and numeric OID

$var.commaoid
>   comma separated numeric OID for array initialization

$var.oidlength
>   length of the oid

$var.subid
>   last number component of oid

$var.module
>   MIB name that the object comes from

$var.parent
>   contains the label of the parent node of $var.

$var.isscalar
> returns 1 if var contains the name of a scalar

$var.iscolumn
> returns 1 if var contains the name of a column

$var.children
> returns 1 if var has children

$var.perltype
> node's perl SYNTAX ($SNMP::MIB{node}{'syntax'})

$var.type
> node's ASN_XXX type (Net-SNMP specific #define)

$var.decl
> C data type (char, u_long, ...)

$var.readable
> 1 if an object is readable, 0 if not

$var.settable
> 1 if an object is writable, 0 if not

$var.creatable
> 1 if a column object can be created as part of a new row, 0 if not

$var.noaccess
> 1 if not-accessible, 0 if not

$var.accessible
> 1 if accessible, 0 if not

$var.storagetype
> 1 if an object is a StorageType object, 0 if not

$var.rowstatus
> 1 if an object is a RowStatus object, 0 if not 'settable', 'creatable', 'lastchange', 'storagetype' and 'rowstatus' can also be used with table variables to indicate whether it contains writable, creatable, LastChange, StorageType or RowStatus column objects

$var.hasdefval
> returns 1 if var has a DEFVAL clause

$var.defval
> node's DEFVAL

$var.hashint
> returns 1 if var has a HINT clause

$var.hint
> node's HINT

$var.ranges
> returns 1 if var has a value range defined

$var.enums
> returns 1 if var has enums defined for it.

$var.access
> node's access type

$var.status
> node's status

$var.syntax
        node's syntax

$var.reference
        node's reference

$var.description
        node's description

SEE ALSO
    mib2c(1)