

SHELL

ΣΗΕΓΓ



November 5, 2025

Canine-Table



POSIX Nexus serves as a comprehensive cross-language reference hub that explores the implementation and behavior of POSIX-compliant functionality across a diverse set of programming environments. Built atop the foundational IEEE Portable Operating System Interface (POSIX) standards, this project emphasizes compatibility, portability, and interoperability between operating systems.

Abstract

Contents

I	IP Module	II
I	Interface Classifier – The Net Glyph	II
I	Layer 2 Address Generator – The L2 Glyph	III
I	Layer 2 Address Inspector – The L2 Audit Glyph	V
I	ARP Table Parser – The ARP Glyph	VI
I	Interface Name Resolver – The Name Glyph	VIII
I	Namespace Lifecycle Manager – The Netns Glyph	X
I	Namespace Executor – The Exec Glyph	XII
I	Altnname Remover – The Alt Glyph	XIII
I	Layer 2 Address Inspector – The L2 Glyph	XIV
I	Interface Name Resolver – The Ifname Glyph	XV
I	Altnname Auditor – The Alt Audit Glyph	XVI

I IP Module

I Interface Classifier – The Net Glyph

nx_ip_net() – Classify Physical and Virtual Interfaces

```

1  nx_ip_net()
2  {
3      ls --color=never -l '/sys/class/net/' | ${AWK:-$(nx_cmd_awk)}
4      -v ex="$1" -F '/' '
5          BEGIN {
6              if (ex == "-e")
7                  ex = "export "
8              else
9                  ex = ""
10             virt = ""
11             phy = ""
12         }
13         /devices\ pci/ {
14             phy = phy " " $NF
15         }
16         /devices\ virtual/ {
17             virt = virt " " $NF
18         }
19     END {
20
21         printf("%sG_NEX_NET_VIRT\x22=%s\x22\n%sG_NEX_NET_PHY=\x22%s\x22\n",
22         ex, substr(virt, 2), ex, substr(phy, 2));
23         '
24     }

```

Interface Classifier – The Net Glyph

- ➔ **Purpose** ↗ Classifies network interfaces into virtual and physical sets using symlink lineage from /sys/class/net
- ➔ **Input** ↗ Optional flag -e to prefix output with export
- ➔ **Mechanism** ↗ AWK parses symlink targets for /devices/pci (physical) and /devices/virtual (virtual)
- ➔ **Output** ↗ Defines G_NEX_NET_VIRT and G_NEX_NET_PHY with space-separated interface names
- ➔ **Use Case** ↗ Used in overlay scripts, interface audits, routing logic, or symbolic network containment



Classify interfaces and export results

- ◁▷ **Initial State** ~ System has lo, eth0, and virbr0
- ◁▷ **Operation** ~ Run nx_ip_net -e to classify and export
- ◁▷ **Expected Result** ~ Outputs export G_NEX_NET_VIRT="lo virbr0" and export G_NEX_NET_PHY="eth0"

```
1 nx_ip_net -e
```

Classify interfaces without export prefix

- ◁▷ **Initial State** ~ System has wlan0 and docker0
- ◁▷ **Operation** ~ Run nx_ip_net without arguments
- ◁▷ **Expected Result** ~ Outputs G_NEX_NET_VIRT="docker0" and G_NEX_NET_PHY="wlan0"

```
1 nx_ip_net
```

I Layer 2 Address Generator – The L2 Glyph

nx_ip_l2() – Generate Unique MAC-like Address

```

1 nx_ip_l2()
2 (
3     eval "$(nx_str_optarg ':n:' '$@')"
4     test -n "$n" && n="-n $n"
5     while : ; do
6         tmpa=$(($AWK:-$(nx_cmd_awk)) -v addr=$(nx_str_rand 12
→xdigit) "
7             BEGIN {
8                 l = split(tolower(addr), hex, "")
9                 do {
10                     s = s ":" hex[1] hex[1-1]
11                 } while ((l-=2) > 0)
12                 delete hex
13                 print substr(s, 2)
14             }
15         ')"
16         g_nx_ip_l2 $n -a | grep -q "$tmpa" || break
17     done

```

```

18     printf '%s\n' "$tmpa"
19 )

```

Layer 2 Address Generator – The L2 Glyph

- ➔ **Purpose** ~> Generates a unique MAC-like address not currently in use
- ➔ **Input** ~> Optional flag -n to namespace the lookup via g_nx_ip_12
- ➔ **Mechanism** ~> Generates 12 random hex digits, formats into colon-separated MAC form, checks for collision
- ➔ **Collision Check** ~> Uses g_nx_ip_12 -a to verify uniqueness before emitting
- ➔ **Use Case** ~> Used in virtual interface creation, container overlays, or symbolic L2 address staging

Generate a unique MAC-like address

- ◀/▶ **Initial State** ~> No arguments passed; default namespace lookup
- ◀/▶ **Operation** ~> Generates and emits a unique address like f a : 3c : 9e : 12 : ab : 77
- ◀/▶ **Expected Result** ~> Returns a MAC-like string not found in g_nx_ip_12 -a

```
1 nx_ip_12
```

Generate address with namespace constraint

- ◀/▶ **Initial State** ~> Namespace veth passed via -n veth
- ◀/▶ **Operation** ~> Ensures uniqueness within g_nx_ip_12 -n veth -a
- ◀/▶ **Expected Result** ~> Returns a MAC-like string not found in the veth namespace

```
1 nx_ip_12 -n veth
```



I Layer 2 Address Inspector – The L2 Audit Glyph

g_nx_ip_l2() – Inspect MAC Addresses Across Namespaces

```

1 g_nx_ip_l2()
2 (
3     eval "$(nx_str_optarg ':n:a' "$@")"
4     test -n "$n" && n="ip netns exec $n "
5     test -n "$a" && $n ip neighbor | ${AWK:-$(nx_cmd_awk)} '{ print
6     →$(NF - 1) }'
7     tmpa=$(($n ip -json address show $NEX_OPT_RMDR 2> /dev/null)
8     →&& nx_data_jdump "$tmpa" | ${AWK:-$(nx_cmd_awk)} '/\.nx
9     \[ [0-9]+\]
10    \.address =/{print $NF} '
11 )

```

Layer 2 Address Inspector – The L2 Audit Glyph

- ➔ **Purpose** ↵ Inspects and emits MAC-like addresses from interfaces and neighbors, optionally scoped to a namespace
- ➔ **Input** ↵ -n for namespace; -a to emit neighbor MACs
- ➔ **Mechanism** ↵ Uses ip -json address show and ip neighbor to extract Layer 2 addresses
- ➔ **Output** ↵ Emits one MAC per line from either interface JSON or neighbor table
- ➔ **Use Case** ↵ Used in collision checks, symbolic L2 audits, or overlay address validation

Emit MAC addresses from current namespace

- ◀/▶ **Initial State** ↵ System has interfaces with MACs like fa:3c:9e:12:ab:77
- ◀/▶ **Operation** ↵ Run g_nx_ip_l2 with no arguments
- ◀/▶ **Expected Result** ↵ Emits MAC-like addresses from ip -json address show

```
1 g_nx_ip_l2
```

Emit neighbor MACs from namespace

- ◁▷ **Initial State** ↵ Namespace veth has active neighbors
- ◁▷ **Operation** ↵ Run g_nx_ip_12 -n veth -a
- ◁▷ **Expected Result** ↵ Emits MACs from ip netns exec veth ip neighbor

```
1 g_nx_ip_12 -n veth -a
```

I ARP Table Parser – The ARP Glyph

nx_ip_arp() – Parse and Emit ARP Table as JSON

```

1 nx_ip_arp()
2 {
3     ${AWK:-$(nx_cmd_awk)} '
4         {
5             if (! header) {
6                 header = 1
7                 next
8             }
9             iface[$NF] = iface[$NF] "{\x22ip\x22:\x22"
10            →$1 "\x22,\x22type\x22:\x22" $2 "\x22,\x22flags\x22:\x22" $3
11            →"\x22,\x22hw\x22:\x22" $4 "\x22,\x22mask\x22:\x22" $5 "\x22},"
12             } END {
13                 for (face in iface) {
14                     sub(/,$/,"]},", iface[face])
15                     s = s "{\x22" face "\x22:\x22"
16                     →iface[face]
17                         }
18             }
19             '$(
20                 test -z "$1" && printf '%s' '/proc/self/net/arp' || {
21                     test -f "$1" && printf '%s'
22                     →"/proc/$1/net/arp" || printf '%s' '/proc/net/arp'
23                     }
24             )

```



ARP Table Parser – The ARP Glyph

- ➔ **Purpose** ~> Parses the system ARP table and emits structured JSON grouped by interface
- ➔ **Input** ~> Optional PID or file path to target a specific ARP table
- ➔ **Mechanism** ~> AWK parses fields from `/proc/* /net/arp` and groups entries by interface
- ➔ **Output** ~> Emits a JSON array of interface-keyed ARP entries with fields: `ip`, `type`, `flags`, `hw`, `mask`
- ➔ **Use Case** ~> Used in symbolic network audits, container overlays, or L2/L3 mapping rituals

Parse current process ARP table

- ◀/▶ **Initial State** ~> System has ARP entries for `eth0` and `docker0`
- ◀/▶ **Operation** ~> Run `nx_ip_arp` with no arguments
- ◀/▶ **Expected Result** ~> Emits JSON with keys `"eth0"` and `"docker0"` containing ARP entries

```
1 nx_ip_arp
```

Parse ARP table for specific PID

- ◀/▶ **Initial State** ~> PID 1234 has a net namespace with ARP entries
- ◀/▶ **Operation** ~> Run `nx_ip_arp 1234`
- ◀/▶ **Expected Result** ~> Emits JSON from `/proc/1234/net/arp` grouped by interface

```
1 nx_ip_arp 1234
```

I Interface Name Resolver – The Name Glyph

nx_ip_name() – Resolve Unique Interface Name

```

1  nx_ip_name()
2  {
3      tmpa="$(
4          eval "$($nx_str_optarg ':n:b:v' '$@')"
5          test -n "$NEX_OPT_RMDR" || exit
6          test -n "$n" && tmpd="ip netns exec $n " || tmpd=""
7          tmpa="${${AWK:-$(nx_cmd_awk)} -v
8          name="$NEX_OPTSTR_RMDR" -v base="$b" 'BEGIN {
9              if (name !~ /^[0-9A-Za-z_-]{1,15}$/)
10                 exit 1
11             if (match(name, /[0-9]+$/)) {
12                 if ((cur = substr(name, 1, RSTART -
13                     ) == base)
14                     exit 3
15                 printf("tmpa=%s tmpb=\x22%s\x22",
16                     substr(name, RSTART), cur)
17                 } else {
18                     printf("tmpa=0 tmpb=\x22%s\x22", name)
19                 }
20             }')" || {
21                 test $? -eq 3 && exit 3
22                 nx_io_printf -E "$NEX_OPT_STR_RMDR is an
23                 invalid name, names must be 1 to 15 character of
24                 '0-9,a-z,A-Z,-._'" 1>&2
25                 unset tmpa
26                 exit 1
27             }
28             eval "$tmpa"
29             while $tmpd ip link show "$tmpb$tmpa" 2>/dev/null
30             >1>&2 || test "$tmpb$tmpa" = "$b"; do
31                 tmpa=$((tmpa+1))
32                 test "${nx_str_len "$tmpb$tmpa")" -le 15 || {
33                     nx_io_printf -E "interface name
34                     '$tmpb$tmpa' is too long, the maximum length is 15." 1>&2
35                     exit 2
36                 }
37                 done
38                 printf 'tmpa=\x22%s\x22 tmpc=\x22%s\x22\n'
39                 "$tmpb$tmpa" "$v"
40             )" || return
41             eval "$tmpa"
42             test -n "$tmpc" && printf '%s\n' "$tmpa"
43             unset tmpc
44         }

```



Interface Name Resolver – The Name Glyph

- ➔ **Purpose** ↵ Resolves a unique interface name by incrementing suffixes and validating namespace collisions
- ➔ **Input** ↵ -n for namespace, -b for base prefix, -v for optional tag
- ➔ **Validation** ↵ Rejects names longer than 15 characters or invalid characters outside [0-9A-Za-z_-]
- ➔ **Mechanism** ↵ Splits numeric suffix, checks for collisions via ip link show, increments until unique
- ➔ **Output** ↵ Emits tmpa as resolved name and tmpc as optional tag
- ➔ **Use Case** ↵ Used in veth pair creation, container overlays, or symbolic interface staging

Resolve unique name with base prefix

- ◀/▶ **Initial State** ↵ Base prefix veth and desired name veth0
- ◀/▶ **Operation** ↵ Run nx_ip_name -b veth veth0
- ◀/▶ **Expected Result** ↵ Emits tmpa="veth1" if veth0 exists

```
1 nx_ip_name -b veth veth0
```

Resolve name within namespace

- ◀/▶ **Initial State** ↵ Namespace ns1 contains eth0
- ◀/▶ **Operation** ↵ Run nx_ip_name -n ns1 eth0
- ◀/▶ **Expected Result** ↵ Emits tmpa="eth1" if eth0 exists in ns1

```
1 nx_ip_name -n ns1 eth0
```

Resolve name with optional tag via -v

- ◁▷ **Initial State** ~ Base prefix veth, desired name veth0, and tag uplink
- ◁▷ **Operation** ~ Run nx_ip_name -b veth -v uplink veth0
- ◁▷ **Expected Result** ~ Emits tmpa="veth1" and tmpc="uplink" if veth0 exists

```
1 nx_ip_name -b veth -v uplink veth0
```

I Namespace Lifecycle Manager – The Netns Glyph

nx_ip_netns() – Create or Remove Network Namespace

```

1 nx_ip_netns()
2 (
3     eval "$(nx_str_optarg 'r' "$@")"
4     eval "$(
5         test -n "$r" && {
6             nx_data_repeat '
7                 ip netns | grep -q "$NEX_ARG" && {
8                     kill "$(cat
9                         →/var/run/nex-$NEX_ARG.pid)" 2> /dev/null
10                    ip netns delete "$NEX_ARG"
11                    rm -f
12                    →"/var/run/netns/nex-$NEX_ARG.pid"
13                    }
14                ' "$NEX_OPT_RMDR"
15            } || {
16                nx_data_repeat '
17                    ip netns | grep -q "$NEX_ARG" || {
18                        ip netns add "$NEX_ARG" && {
19                            ip netns exec
20                            →"$NEX_ARG" sysctl --system 1> /dev/null 2>&1
21                            ip netns exec
22                            →"$NEX_ARG" ip link set lo up
23                            nohup setsid nsenter
24                            →--net="/var/run/netns/$NEX_ARG" sleep infinity 1> /dev/null 2>&1 &
25                            printf $! >
26                            →"/var/run/nex-$NEX_ARG.pid"
27                            }
28                ' "$NEX_OPT_RMDR"
29            }
30        )
31    )
32)

```



Namespace Lifecycle Manager – The Netns Glyph

- ➔ **Purpose** ↵ Creates or removes a Linux network namespace with optional persistent process
- ➔ **Input** ↵ Namespace name via trailing argument; `-r` flag triggers removal
- ➔ **Mechanism** ↵ Uses `ip netns` to create/delete; spawns persistent `sleep` via `nsenter` and stores PID
- ➔ **Output** ↵ Creates or removes `/var/run/nex-$name.pid` and namespace entry in `/var/run/netns`
- ➔ **Use Case** ↵ Used in container overlays, symbolic network isolation, or namespace lifecycle rituals

Create a new network namespace

- ◁/▷ **Initial State** ↵ No namespace named `ns1` exists
- ◁/▷ **Operation** ↵ Run `nx_ip_netns ns1`
- ◁/▷ **Expected Result** ↵ Creates `ns1`, brings up `lo`, spawns persistent `sleep`, stores PID

```
1 nx_ip_netns ns1
```

Remove an existing network namespace

- ◁/▷ **Initial State** ↵ Namespace `ns1` exists with PID file
- ◁/▷ **Operation** ↵ Run `nx_ip_netns -r ns1`
- ◁/▷ **Expected Result** ↵ Kills persistent process, deletes namespace, removes PID file

```
1 nx_ip_netns -r ns1
```

I Namespace Executor – The Exec Glyph

`__nx_ip_exec()` – Emit Namespace Execution Prefix

```

1  __nx_ip_exec()
2  {
3      test -n "$1" && {
4          nx_ip_netns "$1"
5          printf '%s ' "${1:+ip netns exec $1}"
6      }
7 }
```

Namespace Executor – The Exec Glyph

- ➔ **Purpose** ↵ Emits a namespace-prefixed command string for use in subshells or command substitution
- ➔ **Input** ↵ Namespace name as first argument
- ➔ **Mechanism** ↵ Ensures namespace exists via `nx_ip_netns`, then emits `ip netns exec $name`
- ➔ **Output** ↵ Prints `ip netns exec $name` if namespace is provided; empty string otherwise
- ➔ **Use Case** ↵ Used to prefix commands with namespace context in symbolic overlays or interface rituals

Emit namespace exec prefix for a given namespace

- ◁/▷ **Initial State** ↵ Namespace `ns1` may or may not exist
- ◁/▷ **Operation** ↵ Run `__nx_ip_exec ns1`
- ◁/▷ **Expected Result** ↵ Ensures `ns1` exists, emits `ip netns exec ns1`

```
1  __nx_ip_exec ns1
```



I Altnname Remover – The Alt Glyph

d_nx_ip_alt() – Remove Alternate Interface Name

```

1 d_nx_ip_alt()
2 {
3     g_nx_ip_alt $2 | grep -q '^"$1"' || return 1
4     ${_nx_ip_exec "$2") ip link property del dev
5     ↳$(g_nx_ip_ifname "$1" "$2") altnname "$1"
6 }
```

Altnname Remover – The Alt Glyph

- ➔ **Purpose** ↵ Removes an alternate name from a network interface, scoped optionally to a namespace
- ➔ **Input** ↵ \$1: altnname to remove; \$2: optional namespace
- ➔ **Validation** ↵ Checks if altnname exists via g_nx_ip_alt; aborts if not present
- ➔ **Mechanism** ↵ Uses ip link property del to remove altnname from resolved interface name
- ➔ **Use Case** ↵ Used in symbolic interface cleanup, altnname mutation, or namespace-scoped interface audits

Remove altnname from interface in namespace

- ◀/▶ **Initial State** ↵ Interface eth0 in namespace ns1 has altnname uplink
- ◀/▶ **Operation** ↵ Run d_nx_ip_alt uplink ns1
- ◀/▶ **Expected Result** ↵ Removes altnname uplink from eth0 in ns1

```
1 d_nx_ip_alt uplink ns1
```

I Layer 2 Address Inspector – The L2 Glyph

g_nx_ip_l2() – Emit MAC-like Addresses from Namespace

```

1 g_nx_ip_l2()
2 (
3     eval "$nx_str_optarg ':n:a' \"$@\""
4     test -n "$n" && n="ip netns exec $n "
5     test -n "$a" && $n ip neighbor | ${AWK:-$(nx_cmd_awk)} '{
6     print $(NF - 1) '
7     tmpa=$(($n ip -json address show $NEX_OPT_RMDR 2>
8     /dev/null) && nx_data_jdump "$tmpa" | ${AWK:-$(nx_cmd_awk)}
9     '/\.\nx
10    \[ [0-9]+\]
11    \.address =/{print $NF} '
12 )

```

Layer 2 Address Inspector – The L2 Glyph

- ➔ **Purpose** ↵ Emits MAC-like addresses from interfaces and neighbors, optionally scoped to a namespace
- ➔ **Input** ↵ -n for namespace; -a to emit neighbor MACs
- ➔ **Mechanism** ↵ Uses ip -json address show and ip neighbor to extract Layer 2 addresses
- ➔ **Output** ↵ Emits one MAC per line from either interface JSON or neighbor table
- ➔ **Use Case** ↵ Used in collision checks, symbolic L2 audits, or overlay address validation

Emit MAC addresses from current namespace

- ◀/▶ **Initial State** ↵ System has interfaces with MACs like fa:3c:9e:12:ab:77
- ◀/▶ **Operation** ↵ Run g_nx_ip_l2 with no arguments
- ◀/▶ **Expected Result** ↵ Emits MAC-like addresses from ip -json address show

```
1 g_nx_ip_l2
```



Emit neighbor MACs from namespace

- ◁▷ **Initial State** ↵ Namespace ns1 has active neighbors
- ◁▷ **Operation** ↵ Run g_nx_ip_12 -n ns1 -a
- ◁▷ **Expected Result** ↵ Emits MACs from ip netns exec ns1 ip neighbor

```
1 g_nx_ip_12 -n ns1 -a
```

I Interface Name Resolver – The Ifname Glyph

g_nx_ip_ifname() – Resolve Kernel Interface Name

```
1 g_nx_ip_ifname()
2 (
3     tmpa=$(($__nx_ip_exec "$2") ip -json link show $1 2>
4     ↪/dev/null) && nx_data_jdump "$tmpa" | ${AWK:-$(nx_cmd awk)} '/.nx
5     \[ [0-9]+\]
6     \.ifname/{print $NF}
7     '
8
9 )
```

Interface Name Resolver – The Ifname Glyph

- ➔ **Purpose** ↵ Resolves the kernel-assigned interface name for a given link index or identifier
- ➔ **Input** ↵ \$1: link index or name; \$2: optional namespace
- ➔ **Mechanism** ↵ Uses ip -json link show and parses ifname field via nx_data_jdump
- ➔ **Output** ↵ Emits the resolved interface name as seen by the kernel
- ➔ **Use Case** ↵ Used in altname mutation, symbolic interface mapping, or namespace-scoped link audits

Resolve interface name from link index in namespace

- ◁▷ **Initial State** ↵ Link index 3 exists in namespace ns1
- ◁▷ **Operation** ↵ Run `g_nx_ip_ifname 3 ns1`
- ◁▷ **Expected Result** ↵ Emits interface name like `eth0` or `veth3`

```
1 g_nx_ip_ifname 3 ns1
```

I Altnname Auditor – The Alt Audit Glyph

`g_nx_ip_alt()` – Emit Alternate Interface Names

```
1 g_nx_ip_alt()
2 (
3     eval "$(nx_str_optarg ':n:' "$@")"
4     test -n "$n" && n="ip netns exec $n "
5     $n ip -json link show $NEX_OPT_RMDR 2> /dev/null |
6     nx_data_jdump | ${AWK:-$(nx_cmd_awk)} '/\.\altname/{print $NF}'
7 )
```

Altnname Auditor – The Alt Audit Glyph

- ➔ **Purpose** ↵ Emits alternate names assigned to a given interface, optionally scoped to a namespace
- ➔ **Input** ↵ `-n` for namespace; trailing argument is interface name or index
- ➔ **Mechanism** ↵ Uses `ip -json link show` and parses `altname` fields via `nx_data_jdump`
- ➔ **Output** ↵ Emits one altname per line
- ➔ **Use Case** ↵ Used in symbolic interface audits, altname mutation, or namespace-scoped link overlays



Emit altnames from interface in current namespace

- ◁▷ **Initial State** ~ Interface eth0 has altnames uplink and primary
- ◁▷ **Operation** ~ Run g_nx_ip_alt eth0
- ◁▷ **Expected Result** ~ Emits uplink and primary on separate lines

```
1 g_nx_ip_alt eth0
```

Emit altnames from interface in namespace

- ◁▷ **Initial State** ~ Interface veth0 in namespace ns1 has altname peer
- ◁▷ **Operation** ~ Run g_nx_ip_alt -n ns1 veth0
- ◁▷ **Expected Result** ~ Emits peer

```
1 g_nx_ip_alt -n ns1 veth0
```