Raw Sockets and Packet Layouts in Unicon

Jafar Al-Gharaibeh

Unicon Technical Report: 29

July 2026

Abstract

Unicon already opened TCP and UDP sockets through open(), but IP-level protocols such as ICMP and IGMP were not supported. This report describes raw sockets: mode "nr" opens SOCK_RAW, proto= selects the IP protocol number, and writes()/receive() carry upper-layer bytes on the same datagram path UDP already uses. The runtime does not assemble protocol payloads. Package net supplies PacketSpec, a declarative engine for fixed binary layouts, plus ICMP echo and IGMPv2/v3 classes and session wrappers.

Keywords: Unicon, raw sockets, SOCK_RAW, ICMP, IGMP, PacketSpec, runtime, technical report.

Unicon Project
http://unicon.org
University of Idaho
Department of Computer Science
Moscow, ID, 83844, USA

1 1. Introduction

This report describes raw IP sockets in Unicon: how a handle is opened, how datagrams are sent and received, how protocol bytes are built and decoded, and what remains out of scope. It is formatted as a Unicon Technical Report [1]. Socket attributes (`ttl`, `iface`, `join`, and the rest) are the multicast facility [2]; this report adds mode `nr` and the packet library on top of that. The language-facing reference lives in *Programming with Unicon* [3] (Chapter 5, "Raw Sockets" and "PacketSpec and package net"; Chapter 14, `uping` and `uigmp`) and the language reference (`open` mode `nr`, attributes `proto` and `hdrincl`, subscript peek). Automated plumbing and peek tests live in `tests/posix/raw_sock.icn`.

Raw sockets are part of the POSIX network facility. A build without POSIX networking has no `n` modes at all. Opening `SOCK_RAW` usually needs elevated privileges (root or administrator); that is an operating-system restriction, not a Unicon one.

2 2. Motivation

TCP and UDP cover most application protocols. Diagnostic and control traffic sits one layer down: ICMP echo [4], IGMP membership [5] [6], OSPF, GRE, PIM. Those messages are IP protocol payloads, not port-addressed transport streams. ICMP, IGMP, and other raw IP protocols were not supported.

The rest of Unicon networking already treats `open()` as the constructor and `Attrib()` as the option verb. Raw sockets follow that pattern instead of adding `rawsocket()` or a parallel packet API. Payload construction is a library problem: the runtime delivers bytes, and `PacketSpec` names the fields.

3 3. Design principles

`r` after `n` is the socket type, like `u` for UDP. Mode letter `r` remains ordinary read mode unless a preceding `n` has already marked the handle as a network socket. Then `"nr"` means `SOCK_RAW`, the same way `"nu"` means `SOCK_DGRAM`. `"rn"` is not a raw socket: `r` is consumed as read mode before `n` is seen.

`proto=` **is required and is fixed at** `socket()` **time.** The third argument to `socket(2)` is the IP protocol number. There is no useful default: ICMP, IGMP, and OSPF are different sockets. A missing, empty, or unknown `proto=` is error 1310 (`bad socket attribute`). `f["proto"]` peeks the protocol number stored at `open()`; `Attrib()` cannot change it.

The runtime does not build protocol payloads. Writes send whatever bytes the caller supplies. Receives return whatever the kernel delivered, often including an IPv4 header on Linux. `import net` and `PacketSpec` (or hand-assembled strings) fill that gap.

Datagram I/O matches UDP. Raw sockets are not connected. `writes()` uses `sendto(2)` with the destination saved at `open()`. Incoming packets use `receive()`, which returns a record with `addr`, `msg`, `saddr`, `daddr`, and (for IPv4 raw datagrams that include a header) `ttl` and `proto`. Those extra fields are also available as `r.ttl` / `r.saddr`. Use `receive()`, not `read()` / `reads()`.

Two layers for callers. Session wrappers (`Ping`, `Igmp`) hide mode letters for the common cases. PacketSpec classes are there when the program must craft or classify bytes itself.

4 4. Opening a raw socket

```
f := open("8.8.8.8", "nr", "proto=icmp", "ttl=64") |
     stop(&errortext)
```

The first argument is a bare host or address, not `host:port`. Raw destinations are hosts. A single `host:port` form is accepted when there is exactly one colon and a non-empty port, but ICMP and IGMP do not use ports. IPv6 literals contain colons; they are treated as hosts and are not split as `host:port`.

`proto=` takes a symbolic name or an integer 0–255:

Name	Protocol
`icmp`	`IPPROTO_ICMP`

Name	Protocol
`icmpv6` / `icmp6`	`IPPROTO_ICMPV6` (fails if the host has no constant)
`igmp`	`IPPROTO_IGMP`
`tcp` / `udp`	`IPPROTO_TCP` / `IPPROTO_UDP`
`gre`	47
`ospf`	89
`pim`	`IPPROTO_PIM`
`raw`	`IPPROTO_RAW`
`0 .. 255`	that protocol number

Names are case-insensitive. Unknown names and numbers outside 0–255 fail at `open()` with 1310.

If the kernel refuses `SOCK_RAW`, `open()` fails and `&errortext` is the system message (typically "Operation not permitted" or "Permission denied"). That is success of the Unicon plumbing and a privilege problem on the host.

```
procedure main()
   local f
   if f := open("8.8.8.8", "nr") then
      write("oops missing proto")
   else
      write("missing-proto: ", &errortext)
   if f := open("8.8.8.8", "nr", "proto=nope") then
      write("oops bad proto")
   else
      write("bad-proto: ", &errortext)
   if f := open("8.8.8.8", "nr", "proto=icmp") then
      write("raw-icmp: ok")
   else
      write("raw-icmp: ", &errortext)
end
```

Output:

```
missing-proto: bad socket attribute
bad-proto: bad socket attribute
raw-icmp: Operation not permitted
```

The last line is the unprivileged case. With privileges it prints `raw-icmp: ok` (or the test suite's `plumbing-ok`).

5 5. Attributes and status peek

Trailing `open()` arguments are the same `name=value` socket attributes used for TCP and UDP [2]. Booleans are exactly `yes` or `no`. `ttl` and `iface` were named for this sharing: hop limit and interface are not multicast-only knobs.

Attribute	Role on a raw socket
`proto=`	Required. Consumed at `socket()`. Peek with
`hdrincl=yes`	Sets `IP_HDRINCL` so writes may include a complete IPv4
`ttl=`	Unicast and multicast hop limits (`IP_TTL` /
`rcvbuf=` / `sndbuf=`	Socket buffer sizes in bytes.
`broadcast=` / `mcastloop=`	Same meaning as UDP.
`join=` / `leave=` / `source=` / `iface=`	Multicast membership, including SSM `source@group`

`Attrib(f, "ttl=4")` sets hop limits after `open()`. Status uses the same `[ ]` / `key()` get as UDP [2]: unknown names raise 1310, unpopulated fields fail. Boolean peek fields that answered succeed with `"yes"` or `&null` (never `"no"`); that is output only – `open()` and `Attrib()` still take `yes` or `no`. Peeking a closed handle is error 174. `key(f)` that has not yet produced a name also raises 174 if the handle is already closed; after the first suspend, `close()` makes the generator fail instead of raising. Raw sockets add `proto` (fixed at `socket()`) and `hdrincl`.

Name	Peek value
`proto`	IP protocol number from `proto=`: `1` (`icmp`),
`hdrincl`	`"yes"` or `&null` (not `"no"`)
`ttl`	Hop limit (integer), e.g. `64`
`iface`	Outbound / join interface, e.g. `127.0.0.1`
`groups`	List of joined groups, e.g. `239.1.1.1`; unpopulated
`mcastloop` / `reuseaddr` / `reuseport` /	`"yes"` or `&null` (not `"no"`)
`rcvbuf` / `sndbuf`	Buffer sizes in bytes (integers)

```
f := open("127.0.0.1", "nr", "proto=icmp") | stop(&errortext)
write("proto=", f["proto"])
write("ttl=", f["ttl"] | "unpopulated")
every k := key(f) do
   write("field ", k)
close(f)
```

6 6. Send and receive

```
writes(f, icmp_bytes)
r := receive(f)
write(r.addr, "  ", *r.msg, " bytes")
```

`writes()` calls `sock_write()`, which for `SOCK_RAW` (and UDP) does `sendto` to the `addrinfo` saved at `open()`. There is no `connect(2)`. Changing the destination means opening another handle (or, with `hdrincl=yes`, putting the destination in a supplied IP header).

`receive()` peeks with `MSG_PEEK` into a 2K buffer, then consumes with `recvfrom`. If the peek fills 2K, the consume uses a 64K buffer so a large datagram is not truncated. The result is a `posix_message`: `addr` is the peer (often `a.b.c.d:0` for raw IPv4), `msg` is the byte string. `saddr` / `daddr` use the usual IP-header names: on an IPv4 raw receive they are the header source and destination; on UDP, `saddr` is the numeric peer address and `daddr` is unpopulated. When the receive buffer starts with an IPv4 header, `ttl` and `proto` are that header's hop limit and protocol; otherwise those fields are null.

On many IPv4 stacks a raw receive buffer starts with the IP header, then the protocol message. Linux ICMP is the usual example. Strip it before decoding ICMP or IGMP (section 7). IPv6 raw sockets typically deliver the upper layer only; there is no IPv6 header-strip helper yet.

`select()` works on the handle. Session wrappers use it to implement timeouts around `receive()`.

As with UDP, `read()` / `reads()` are the wrong verbs for datagrams. Use `receive()`.

7 7. PacketSpec

Module `packetspec` (package `net`, `uni/lib/packetspec.icn`) is a small declarative engine for fixed binary layouts. Protocol classes inherit `PacketSpec` and override `define()`.

Layer-1 helpers pack and unpack network-order integers (`u8` / `u16` / `u32` / `u64` and the matching `get_u*` readers) and IPv4 addresses (`ipv4` / `to_ipv4` / `get_ipv4`). `inet_cksum()` is RFC 1071 [7].

7.1 7.1 Declaring a layout

```
class IcmpEchoRequest : IcmpPacket()
   method define()
      field("type", 1, 8)
      field("code", 1, 0)
      checksum_field("checksum")
      field("id", 2)
```

```
      field("seq", 2)
   end
end
```

`field(name, size, defval)` is a network-order integer. `bytes_field` is opaque fixed-width bytes. `ipv4_field` accepts a dotted-quad, integer, or four raw bytes on build and returns a dotted-quad on decode. `checksum_field` is a 16-bit placeholder filled at `build()` time over the header plus payload. Pass `exclude_off` / `exclude_len` (1-based) to omit a range from the sum — OSPF Authentication and PIM Register are the documented cases; those layouts are not in the library yet.

`require(name)` marks a field that `build()` must see via `set()` or the optional extra table. `relax()` / `build(..., lax)` skip that check for incomplete test packets. `strict()` and `clear()` turn checking back on.

7.2 7.2 Build, decode, describe

Preferred call shape (no string-keyed tables required). This needs no raw socket:

```
import net

procedure main()
   local pkt, v
   write(IcmpEchoRequest().describe())
   write(IcmpEchoRequest().set("id", 1).set("seq", 2).describe())
   pkt := IcmpEchoRequest().set("id", 1).set("seq", 2).build(, "hi")
   write("len=", *pkt)
   v := IcmpEchoRequest().decode(pkt)
   write("type=", v["type"], " id=", v["id"], " seq=", v["seq"])
end
```

Output:

```
type:int=8  code:int=0  checksum:checksum(auto)  id:int=0  seq:int=0
type:int=8  code:int=0  checksum:checksum(auto)  id:int=1  seq:int=2
len=10
type=8 id=1 seq=2
```

`set()` and `set_payload()` are chainable. `build(extra, payload, lax)` overlays an optional table, appends payload, and returns the complete byte string with checksum filled. `decode(s)` returns a table of field names to values. `describe()` is a one-line summary for the REPL: names, kinds, defaults, required markers, and current `set()` values.

7.3 7.3 IPv4 receive helpers

`strip_ipv4(pkt)` uses the header IHL to return the upper-layer message. `ipv4_ttl(pkt)` and `ipv4_src(pkt)` read TTL and source from the same leading header. ICMP and IGMP classes call `strip_ipv4` before matching.

8 8. ICMP echo

`uni/lib/icmp.icn` adds echo request (type 8) and echo reply (type 0) [4] on top of `PacketSpec`:

```
PacketSpec
  IcmpPacket          strip_ip / match
    IcmpEchoRequest   type 8; echo(id, seq, payload)
    IcmpEchoReply     type 0; match_echo(pkt, id, seq)
Ping                  session wrapper
```

`Ping(host)` opens `proto=icmp` with TTL 64 and a 2000 ms default wait. `echo()` sends a request (56 zero bytes unless a payload is given), waits with `select()`, and succeeds with a decode table plus `rtt` (milliseconds,

real), `rtt_us`, `ttl` (from the IPv4 header), and `addr`. Round-trip time uses `gettimeofday()`.

```
import net

procedure main()
   local p, v
   p := Ping("8.8.8.8") | stop(&errortext)
   write(p.describe())
   if v := p.echo() then
      write("rtt=", v["rtt"], " from ", v["addr"])
   p.close()
end
```

`set_timeout(ms)` changes the default wait. An explicit sequence number and payload can be passed to `echo()`; otherwise the session increments `seq`.

The same exchange with an explicit socket:

```
import net

procedure main()
   local f, req, reply, id, pkt, r
   req := IcmpEchoRequest()
   reply := IcmpEchoReply()
   id := iand(?100000 | 1, 16rFFFF)
   f := open("8.8.8.8", "nr", "proto=icmp", "ttl=64") |
        stop(&errortext)
   pkt := req.set("id", id).set("seq", 1).build(, "unicon")
   writes(f, pkt)
   r := receive(f)
   if reply.match_echo(r.msg, id, 1) then
      write("reply from ", r.addr)
   close(f)
end
```

Other ICMP types (destination unreachable, time exceeded, and so on) are not in the library. `proto=icmpv6` opens an ICMPv6 socket, but there are no Neighbor Discovery or echo layouts yet.

9 9. IGMP

`uni/lib/igmp.icn` covers IGMPv2 (RFC 2236) and IGMPv3 (RFC 3376):

```
PacketSpec
  IgmpPacket
    Igmpv2Query / Igmpv2Report / Igmpv2Leave
    Igmpv3Query / Igmpv3Report
Igmp                      session wrapper
```

IGMPv2 messages are an eight-octet header. Report and leave `require()` a group. IGMPv3 queries add flags, QQIC, and a source list. IGMPv3 reports carry group records: ASM join/leave and SSM source filters (`join_sources`, `allow`, `block`).

`Igmp()` opens `proto=igmp` bound at `0.0.0.0`. `recv(ms)` waits, then returns a table with `kind`, `addr`, `msg`, `len`, and decoded fields when recognized. `kind` is one of `v3_report`, `v3_query`, `v2_query`, `v2_report`, `v2_leave`, or `unknown`.

PacketSpec classes work without a raw socket:

```
import net

procedure main()
   local pkt, v
   write(Igmpv2Report().describe())
   write(Igmpv3Report().describe())
   pkt := Igmpv2Report().report("239.1.1.1")
   write("v2 report len=", *pkt)
   v := Igmpv2Report().decode(pkt)
```

```
   write("type=", v["type"], " group=", v["group"])
   pkt := Igmpv3Report().join("239.1.1.1")
   write("v3 join len=", *pkt)
end
```

Output:

```
type:int=22  max_resp_time:int=0  checksum:checksum(auto)
↪  group:ipv4(required)
type:int=34  reserved1:int=0  checksum:checksum(auto)
↪  reserved2:int=0  nrecords:int=0
v2 report len=8
type=22 group=239.1.1.1
v3 join len=16
```

Opening a live IGMP socket needs the same privileges as ICMP:

```
import net

procedure main()
   local g, v
   g := Igmp() | stop(&errortext)
   write(g.describe())
   if v := g.recv(1000) then
      write(v["kind"], " from ", v["addr"])
   g.report("239.1.1.1")    # IGMPv2 membership report
   g.join("239.1.1.1")      # IGMPv3 ASM join
   g.close()
end
```

Session send helpers: `report` / `leave` / `query` (v2), `join` / `leave3` (v3 ASM), and `send(bytes)` for a hand-built packet. SSM source-filter joins use the PacketSpec class, not the session wrapper:

```
pkt := Igmpv3Report().join_sources("239.1.1.1", ["192.0.2.1"])
g.send(pkt)
```

Exported constants (`IGMP_MEMBERSHIP_QUERY`, `IGMP_V3_MEMBERSHIP_REPORT`, `IGMP_CHANGE_TO_EXCLUDE_MODE`, and the rest) are initialized on first `IgmpPacket` construction.

10 10. Sample programs

`uni/progs/uping` and `uni/progs/uigmp` are teaching clients, not replacements for `ping(8)` or `tcpdump(1)`. They need `SOCK_RAW` privileges:

```
unicon uping
unicon uigmp
sudo ./uping host [count]
sudo ./uigmp [seconds]
sudo ./uping -v host    # also show open()/Ping API demos
sudo ./uigmp -v         # also show open()/Igmp API demos
```

Default mode uses the session wrappers. `-v` / `-d` also runs the lower-level `open(..., "nr", ...)` plus PacketSpec path and prints `describe()` output.

```
sudo ./uping 192.0.2.1
```

Output:

```
PING 192.0.2.1 (192.0.2.1): 56 data bytes
64 bytes from 192.0.2.1: icmp_seq=0 ttl=118 time=18.390 ms
64 bytes from 192.0.2.1: icmp_seq=1 ttl=118 time=16.245 ms
64 bytes from 192.0.2.1: icmp_seq=2 ttl=118 time=15.533 ms
64 bytes from 192.0.2.1: icmp_seq=3 ttl=118 time=17.311 ms
--- 192.0.2.1 ping statistics ---
4 packets transmitted, 4 packets received, 0.0% packet loss
round-trip min/avg/max = 15.533/16.869/18.390 ms
```

```
sudo ./uigmp
```

Output:

```
listening on IGMP

36 bytes from 192.0.2.10: IGMPv3 query group=0.0.0.0 max_resp=100
↪  qrv=2
36 bytes from 192.0.2.10: IGMPv3 query group=239.1.1.1 max_resp=100
↪  qrv=2
36 bytes from 192.0.2.20: IGMPv3 report {239.1.1.1
↪  CHANGE_TO_EXCLUDE}
```

11 11. Runtime implementation

The work is in the existing POSIX socket code, not a new file type.

Mode letter. `src/runtime/fsys.r`: after `n` has set `Fs_Socket`, `r` / `R` sets `sock_type = SOCK_T_RAW` and does not set the ordinary read bit. Without `Fs_Socket`, `r` remains `Fs_Read`. `e` (crypto raw-key material) is a later check on the same letter; `"nr"` is a socket, `"er"` is a crypto handle.

Create. `src/runtime/rposix.r`: `sock_connect` (and the listen/bind path) require `proto=` before `socket()`. `uni_getaddrinfo` allows a missing port for `SOCK_T_RAW`. The real protocol number is passed as the third argument to `socket()`; `getaddrinfo` is told protocol 0 so it does not override it.

I/O. UDP and raw share `saddrs[]`: the `addrinfo` from `open()` is kept until `sock_close()`. `sock_write` `sendto`s there. `sock_recv` accepts `SOCK_DGRAM` and `SOCK_RAW`. `proto=` is skipped in `setsockopt` (already consumed). `hdrincl=` sets `IP_HDRINCL`.

Errors. Unknown or missing socket attributes use error 1310 with the offending string (`proto`, `proto=nope`, `hdrincl=maybe`). Kernel failures use the system `&errortext`.

There is no `&features` flag named "raw sockets". If POSIX sockets exist, mode `nr` exists.

12 12. Status and remaining work

The language surface described here is implemented: `"nr"`, required `proto=`, `hdrincl=`, datagram `writes` / `receive`, PacketSpec, ICMP echo, IGMPv2/v3, `uping`, and `uigmp`.

More PacketSpec layouts. GRE, OSPF, and PIM names are accepted at `socket()` and the checksum helper documents their exclude ranges, but there are no protocol classes yet.

ICMPv6. `proto=icmpv6` opens the socket. Echo, Neighbor Discovery, and an IPv6 header strip are not in `net`.

IPv6 receive metadata. `strip_ipv4` / `ipv4_ttl` assume a leading IPv4 header. IPv6 raw receives need their own helpers.

`read()` on raw handles. UDP clears the read bit so `read()` fails. Raw currently keeps `Fs_Read` because it shares the TCP-like status assignment. The documented API is still `receive()`. Aligning the status bits with UDP would make misuse fail earlier.

Privileged tests. `tests/posix/raw_sock.icn` checks plumbing without sending packets. A live ICMP echo test would need root in CI and a reachable responder; it is not in the suite.

Windows. Opening `SOCK_RAW` is possible with administrator rights, but Windows historically restricts which protocols a raw socket may use. Treat `uping` / `uigmp` as UNIX-first tools.

Header inclusion. `hdrincl=yes` is wired. There is no PacketSpec IPv4 header class to go with it; callers who craft full datagrams build those bytes themselves.

13 Appendix: test coverage

`tests/posix/raw_sock.icn` does not require `SOCK_RAW` success. It checks that:

- missing, empty, and unknown `proto=` fail with 1310
- `proto=icmp`, `gre`, `ospf`, and numeric `89` reach `socket()` (success or a privilege denial both print `plumbing-ok`)
- `hdrincl=maybe` is rejected
- without a preceding `n`, mode `"r"` still opens a file for reading
- when `SOCK_RAW` is allowed, `f["proto"]` is 1 after `proto=icmp` and `key(f)` yields populated names (privilege denial still prints `raw-peek:ok` / `raw-key:ok` so the `.std` stays portable)

Expected output is `tests/posix/stand/raw_sock.std`:

```
missing-proto: bad socket attribute
bad-proto: bad socket attribute
empty-proto: bad socket attribute
raw-icmp: plumbing-ok
raw-gre: plumbing-ok
raw-ospf: plumbing-ok
raw-numeric: plumbing-ok
bad-hdrincl: bad socket attribute
read-mode-r: ok
raw-peek:ok
raw-key:ok
```

On an unprivileged account the `raw-*` lines still print `plumbing-ok`: the test treats both a successful `socket()` and a privilege denial as passing plumbing. The posix suite picks up every `*.icn` in that directory.

14 References

References

[1] Clinton Jeffery. "How to Write a Unicon Technical Report". doc/utr/utr15.tex. 2013.

[2] Jafar Al-Gharaibeh. "Multicast and Socket Attributes in Unicon". doc/utr/utr28.rst. 2026.

[3] Clinton Jeffery and Shamim Mohamed and Jafar Al-Gharaibeh. "Programming with Unicon". doc/book/. 2026.

[4] J. Postel. "Internet Control Message Protocol". RFC 792. 1981.

[5] W. Fenner. "Internet Group Management Protocol, Version 2". RFC 2236. 1997.

[6] B. Cain and S. Deering and I. Kouvelas and B. Fenner and A. Thyagarajan. "Internet Group Management Protocol, Version 3". RFC 3376. 2002.

[7] R. Braden and D. Borman and C. Partridge. "Computing the Internet Checksum". RFC 1071. 1988.