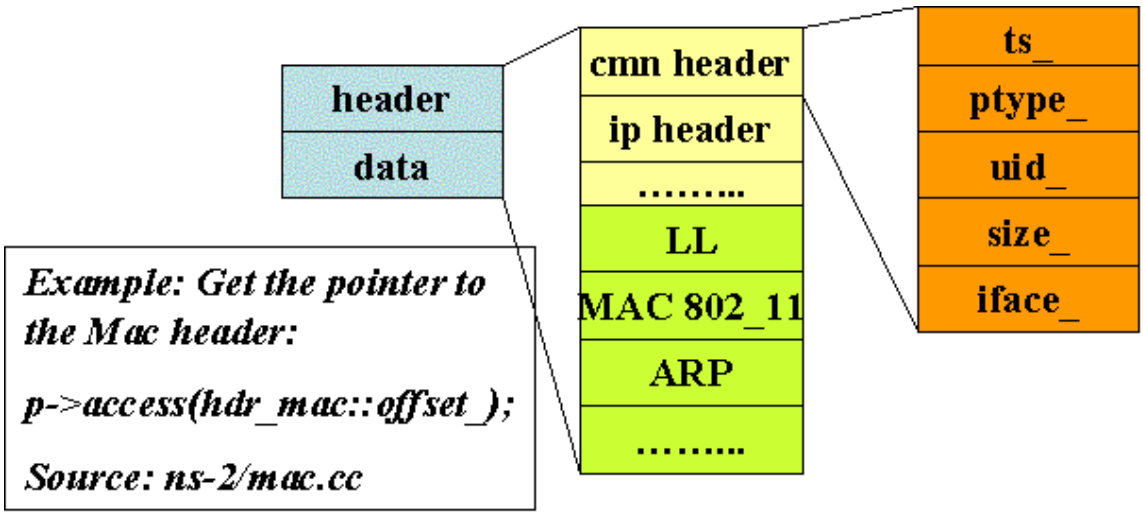
Headers & Addresses in ns-2

[Common | 802.11 MAC | IP | DSR | Back to *Network Simulator 2 for Wireless*]

Notice that by default, all packet headers are included.

A diagram:



Common-header:
Access method:

```
struct hdr_cmn {
        enum dir_t { DOWN= -1, NONE= 0, UP= 1 };
        packet_t ptype_;        // packet type (see above)
        int     size_;          // simulated packet size
        int     uid_;           // unique id
        int     error_;         // error flag
        int     errbitcnt_;     // # of corrupted bits jahn
        int     fecsize_;
        double  ts_;            // timestamp: for q-delay measurement
        int     iface_;         // receiving interface (label)
        dir_t   direction_;     // direction: 0=none, 1=up, -1=down
        // source routing
        char src_rt_valid;

        //Monarch extn begins
        nsaddr_t prev_hop_;     // IP addr of forwarding hop
        nsaddr_t next_hop_;     // next hop for this packet
        int      addr_type_;    // type of next_hop_ addr
        nsaddr_t last_hop_;     // for tracing on multi-user channels

        // called if pkt can't obtain media or isn't ack'd. not called if
        // droped by a queue
        FailureCallback xmit_failure_;
        void *xmit_failure_data_;

        /*
         * MONARCH wants to know if the MAC layer is passing this back because
         * it could not get the RTS through or because it did not receive
```

```
         * an ACK.
         */
        int     xmit_reason_;

#define XMIT_REASON_RTS 0x01
#define XMIT_REASON_ACK 0x02

        // filled in by GOD on first transmission, used for trace analysis
        int num_forwards_;      // how many times this pkt was forwarded
        int opt_num_forwards_;   // optimal #forwards
        // Monarch extn ends;

        // tx time for this packet in sec
        double txtime_;
        inline double& txtime() { return(txtime_); }

        static int offset_;     // offset for this header
        inline static int& offset() { return offset_; }
        inline static hdr_cmn* access(const Packet* p) {
                return (hdr_cmn*) p->access(offset_);
        }

       .....
}
```

`hdr_cmn *ch= hdr_cmn::access(pkt); //common header`

What's in common header (refer to ./common/packet.h)

- ch->ptype() : protocl type. e.g. PT_DSR shows this is a **signaling** message for DSR protocol, not a normal DATA packet
- ch->size();
- ch->direction() : Up or Down
- ch->uid()
- ch->iface(): Interface
- ch->next_hop(): next hop for this packet. IP address in format nsaddr_t*. So, it is amazing to see some codes like, ch->next_hop=MAC_BRPADCST . MAC_BROADCST is defined as #define MAC_BROADCAST ((u_int32_t) 0xffffffff) and nsaddr_t is also deined as int32_t in config.h (typedef int32_t nsaddr_t;). Basically, ns2 use 32bit addressing support.

MAC Ethernet Address in 802.11 MAC Header:
Access method:

```
 struct hdr_mac802_11 {
        struct frame_control    dh_fc;                                  // 2 byte
        u_int16_t               dh_duration;                            // 2
        u_char                  dh_da[ETHER_ADDR_LEN];                  // 6
        u_char                  dh_sa[ETHER_ADDR_LEN];                  // 6
        u_char                  dh_bssid[ETHER_ADDR_LEN];               // 6
        u_int16_t               dh_scontrol;                            // 2
        u_char                  dh_body[0]; // XXX Non-ANSI             // 1   //not allocate, the
};
 .....

 u_int32_t src,dst;
```

```
 struct hdr_mac802_11 *dh = HDR_MAC802_11(p);

//method to get a 32bit general descriptor of address
 dst = ETHER_ADDR(dh->dh_ra);
 src = ETHER_ADDR(dh->dh_ta);

//method to set ....
      int src, dst;
      STORE4BYTE(&dst, (dh->dh_ra));
      STORE4BYTE(&src, (dh->dh_ta));
```

Ethernet Address is specified in mac.h as (#define ETHER_ADDR_LEN 6). Thus, dh_ra and dh_ta are both six-unsinged-character array. Also, there is another defintion for this operation: #define ETHER_ADDR(x) (GET4BYTE(x)), so only the last 4 bytes are reassembled for a 32-bit address.
How MAC address is set? The addr() function is defined in the Class MAC, and the index_ is corresponding internal member (type is integer).

```
static int MacIndex = 0;

Mac::Mac() :
        BiConnector(), abstract_(0), netif_(0), tap_(0), ll_(0), channel_(0), callback_(0),
        hRes_(this), hSend_(this), state_(MAC_IDLE), pktRx_(0), pktTx_(0)
{
        index_ = MacIndex++;
        bind_bw("bandwidth_", &bandwidth_);
        bind_time("delay_", &delay_);
        bind_bool("abstract_", &abstract_);
}

class Mac : public BiConnector {
public:
   ....
   inline int addr() { return index_; }
protected:
   ....
   int index_;             // MAC address
   ....

}

//alos, the  mac-802.11 use index as its mac address to form MAC frames: (see in mac-802_11.cc)
....
STORE4BYTE(&index_, (rf->rf_ta));
```

Space Sharing in MAC-802.11 header

As in packet.h

```
#define HDR_MAC802_11(p) ((hdr_mac802_11 *)hdr_mac::access(p))
```

There is no define of access function of struct hdr_mac802_11. The problem is that, we acccess the header use the struct mac_hdr, and then convert it to a 802.11 header. Thus,

the frame space and frame format should have some realtion:
See.

size of mac header is 36 bytes, but size of mac-802.11 header in ns-2 is only 24 bytes.

IP Addr in IP header
Access Method: (refer to *./common/ip.h*)

```
 struct hdr_ip {
        /* common to IPv{4,6} */
        ns_addr_t       src_;
        ns_addr_t       dst_;
        int             ttl_;

        /* Monarch extn */
//      u_int16_t       sport_;
//      u_int16_t       dport_;

        /* IPv6 */
        int             fid_;   /* flow id */
        int             prio_;

        static int offset_;
        inline static int& offset() { return offset_; }
        inline static hdr_ip* access(const Packet* p) {
                return (hdr_ip*) p->access(offset_);
        }

    /* per-field member acces functions */
        ns_addr_t& src() { return (src_); }
        nsaddr_t& saddr() { return (src_.addr_); }
        int32_t& sport() { return src_.port_;}

        ns_addr_t& dst() { return (dst_); }
        nsaddr_t& daddr() { return (dst_.addr_); }
        int32_t& dport() { return dst_.port_;}
        int& ttl() { return (ttl_); }
        /* ipv6 fields */
        int& flowid() { return (fid_); }
        int& prio() { return (prio_); }
};

......

struct ns_addr_t {
        int32_t addr_;
        int32_t port_;
#ifdef __cplusplus p.dest = ID((Address::instance().get_nodeaddr(iph->daddr())),::IP);
  p.src = ID((
        ns_addr_t& operator= (const ns_addr_t& n) {
                addr_ = n.addr_;
                port_ = n.port_;
                return (*this);
        }
        int operator== (const ns_addr_t& n) {
                return ((addr_ == n.addr_) && (port_ == n.port_));
        } p.dest = ID((Address::instance().get_nodeaddr(iph->daddr())),::IP);
  p.src = ID((
#endif // __cplusplus
```

```
};


hdr_ip *iph =  hdr_ip::access(packet);

//method to set address
  iph->saddr() = Address::instance().create_ipaddr(p.src.addr, RT_PORT); p.dest = ID((Address::instance(
  p.src = ID((
  iph->sport() = RT_PORT;
  iph->daddr() = Address::instance().create_ipaddr(p.dest.addr, RT_PORT);
  iph->dport() = RT_PORT;
  iph->ttl() = 255;
// method to read address ??
   iph->saddr()
   iph->daddr()
   p.dest = ID((Address::instance().get_nodeaddr(iph->daddr())),::IP);
   p.src = ID((Address::instance().get_nodeaddr(iph->saddr())),::IP);
```

As seen, ns_addr_t is different from nsaddr_t, it has ip address and port infomation. Just 32 bit is suitable for ip address.

Very Important! All things above IP (Routing, TCP are) are implemented as Agent. Any agent's message must include an IP header.
The class Agent has functions like addr() and daddr() to get the address of IP header.

```
 class Agent : public Connector {
 public:
    ....
        inline nsaddr_t& addr() { return here_.addr_; }
        inline nsaddr_t& port() { return here_.port_; }
        inline nsaddr_t& daddr() { return dst_.addr_; }
        inline nsaddr_t& dport() { return dst_.port_; }
  protected:
        ...
        ns_addr_t here_;                // address of this agent
        ns_addr_t dst_;                 // destination address for pkt flow
        ...
}


int
Agent::delay_bind_dispatch(const char *varName, const char *localName, TclObject *tracer)
{
        if (delay_bind(varName, localName, "agent_addr_", (int*)&(here_.addr_), tracer)) return TCL_OK;
        if (delay_bind(varName, localName, "agent_port_", (int*)&(here_.port_), tracer)) return TCL_OK;
        if (delay_bind(varName, localName, "dst_addr_", (int*)&(dst_.addr_), tracer)) return TCL_OK;
        if (delay_bind(varName, localName, "dst_port_", (int*)&(dst_.port_), tracer)) return TCL_OK;

......
```

Class Hierarchy : **TclObject ---> NSObject ----> Agent**

DSR's SR (Source Route) Header
the SR has a path informaton from source to a destination, an SR should have following basic information for a baseline version of DSR protocol:

- a path of IP or other addressses

- a type to show this is route-request, route-reply or route-error
- an ID to distinguish duplicate requests.

```
struct sr_addr {
        int addr_type;          /* same as hdr_cmn in packet.h */
        nsaddr_t addr;

        /*
         * Metrics that I want to collect at each node
         */
        double  Pt_;
};

struct route_request {
        int     req_valid_;     /* request header is valid? */
        int     req_id_;        /* unique request identifier */
        int     req_ttl_;       /* max propagation */
};

struct route_reply {
        int     rep_valid_;     /* reply header is valid? */
        int     rep_rtlen_;     /* # hops in route reply */
        struct sr_addr  rep_addrs_[MAX_SR_LEN];
};

struct route_error {
        int     err_valid_;     /* error header is valid? */
        int     err_count_;     /* number of route errors */
        struct link_down err_links_[MAX_ROUTE_ERRORS];
};

.......
class hdr_sr {
private:
        int valid_;             /* is this header actually in the packet?
                                   and initialized? */
        int salvaged_;  /* packet has been salvaged? */

        int num_addrs_;
        int cur_addr_;
        struct sr_addr addrs_[MAX_SR_LEN];

        struct route_request    sr_request_;
        struct route_reply      sr_reply_;
        struct route_error      sr_error_;
...

         inline struct sr_addr* addrs() { return addrs_; }
...
}

//Constructor of Path, convert a SR header into a PATH object
Path::Path(struct hdr_sr *srh)
{ /* make a path from the bits of an NS source route header */
        path = new ID[MAX_SR_LEN];

        if (! srh->valid()) {
                len = 0;
                cur_index = 0;
                return;
        }
```

```
        len = srh->num_addrs();
        cur_index = srh->cur_addr();

        assert(len <= MAX_SR_LEN);

        for (int i = 0 ; i < len ; i++)
                path[i] = ID(srh->addrs()[i]);  //note : both type and addr are copied
}


hdr_sr *srh =  hdr_sr::access(packet);
SRPacket p(packet, srh);
p.dest = ID((Address::instance().get_nodeaddr(iph->daddr())),::IP);
p.src = ID((Address::instance().get_nodeaddr(iph->saddr())),::IP);
```

To get and set SR header, first we need to turn things into a packet of type SRPacket, in this class, route[n] is use to access ID[n], for each ID[n], use inline nsaddr_t getNSAddr_t() to get this addr;
Normally, all SR-related operation are use "ID" not direct 32bit integer to refer.
How the address get set at the very beginning of the DSR Agent, use those :

```
SRNodeNew instproc init args {
 ......
 $dsr_agent_ addr $address_
 ......

}
```