GTP Tunnel Encapsulation Example

The following example demonstrates GTPv1 encapsulation which must be declared as a custom header:

```
header gtp_v1_h {
    bit<3>        version;                  /** For GTPv1, this has a value of 1. */
    bit           protocol_type;            /** GTP (value 1) from GTP' (value 0) */
    bit           reserved;
    bit           extension_header_flag;    /** extension header optional field. */
    bit           seq_number_flag;          /** Sequence Number optional field */
    bit           n_pdu_number_flag;        /** N-PDU number optional field */
    bit<8>        message_type;             /** types of messages are defined in
3GPP TS 29.060 section 7.1 */
    bit<16>       message_length;           /** length of the payload in bytes */
    bit<32>       teid;                     /** Tunnel endpoint identifier */
    bit<16>       sequence_number;          /** optional */
    bit<8>        n_pdu_number;             /** optional */
    bit<8>        next_extension_hdr_type;  /** optional if any of the E, S, or PN
bits are on. The field must be interpreted only if the E bit is on */
}
```

The custom GTPv1 header can optionally be added to the parse graph (see Custom Parser States) is optional, and is only required if any of the custom header fields need to be read from or modified after encapsulation.

```
struct headers_t {
    NV_FIXED_HEADERS
    gtp_v1_h   gtpv1;
}

parser packet_parser(packet_in packet, out headers_t headers) {
    NV_FIXED_PARSER(packet, headers)

    @nv_transition_from("nv_parse_udp", GTP_U_PORT)
    state parse_gtp
```

```
    {
        packet.extract(headers.gtpv1);
        transition nv_parse_inner_ipv4;
    }
}
```

The extern object NvTunnelTemplate<HEADER_TYPE> requires a struct type, for which the entire underlay header is defined as a struct:

```
struct tunnel_headers_t {
    nv_ethernet_h ethernet;
    nv_ipv4_h     ipv4;
    nv_udp_h      udp;
    gtp_v1_h      gtpv1;
}
```

Finally, when declaring a NvTunnelTemplate object, the annotation `@nv_tunnel_fields` must be present. This annotation contains a key-value entry for each header in the specified struct type. Under each header, each header field must be specified in the order that the field appears in the header. For the GTPv1 example, a NvTunnelTemplate would be declared in the main control:

```
@nv_tunnel_fields(
    ethernet = {
        dst_addr = "variable",
        src_addr = "variable",
        ether_type = 0x0800
    },
    ipv4 = {
        version = 0x4,
        ihl = 0x5,
        diffserv = 0,
        ecn = "ignore",                    // Cannot set
```

```
        total_len = "ignore",          // reparse will calculate, cannot set
        identification = "ignore",  // Cannot set
        flags = 0,
        frag_offset = 0,
        ttl = 64,
        protocol = 17,
        hdr_checksum = "ignore",      // reparse will calculate, cannot set
        src_addr = "variable",
        dst_addr = "variable"
    },
    udp = {
        src_port = "ignore",          // reparse will set, cannot be set manually
        dst_port = "variable",
        length   = "ignore",          // reparse will calculate, cannot set
        checksum = "ignore"           // Cannot set
    },
    gtpv1 = {
        version = 1,
        protocol_type = 1,
        reserved = 0,
        extension_header_flag = 1,
        seq_number_flag = 1,
        n_pdu_number_flag = 1,
        message_type = 0xFF,     // T-PDU
        message_length = "variable",
        teid = "variable",
        sequence_number = 0,
        n_pdu_number = 0,
        next_extension_hdr_type = 0
    }
)
NvTunnelTemplate<tunnel_headers_t>() gtpv1Tunnel;
```

The type `tunnel_headers_t` determines the required structure of the annotation. Fields marked "variable" will be determined according to the arguments passed to

`nv_set_l2tunnel_underlay` or `nv_set_l3tunnel_underlay`. Some fields will always be overwritten by reparse, such as IPv4 length, IPv4 checksum, UDP length, etc which must be calculated and cannot be set by the user. Such fields can be marked "ignore". Marking other fields "ignore", would set them to zero.

Finally, the instantiated NvTunnelTemplate object can be used in one of the custom tunnel externs. For example:

```
action forward(nv_ipv4_addr_t src_addr) {
    nv_set_l3tunnel_underlay(headers, gtpv1Tunnel, {
        48w0x001A2B3C4D5E,  // ethernet.dst_addr
        48w0x00F1E2D3C4B5,  // ethernet.src_addr
        src_addr,           // ipv4.src_addr
        32w0xC07B7B01,      // ipv4.dst_addr
        16w0x868,           // udp.dst_port for GTP-U
        16w0x4,             // gtpv1.message_length
        32w0x1234567        // gtpv1.teid
    });
    nv_send_to_port(1);
}
```

The third argument to `nv_set_l3tunnel_underlay` is required to be a list expression where each element corresponds to a header field marked "variable" in the NvTunnelTemplate annotation. The order of the values in the list is the order in which the header fields appear in the packet.

Full example of gtp_tunnel.p4:

```
/*
 * SPDX-FileCopyrightText: Copyright (c) 2025 NVIDIA CORPORATION & AFFILIATES. All rights reserved.
 * SPDX-License-Identifier: LicenseRef-NvidiaProprietary
 *
 * NVIDIA CORPORATION, its affiliates and licensors retain all intellectual
 * property and proprietary rights in and to this material, related
 * documentation and any modifications thereto. Any use, reproduction,
 * disclosure or distribution of this material and related documentation
 * without an express license agreement from NVIDIA CORPORATION or
```

```
 * its affiliates is strictly prohibited.
 */

#include <doca_model.p4>
#include <doca_headers.p4>
#include <doca_externs.p4>
#include <doca_parser.p4>

#define GTP_U_PORT 2152

header gtp_v1_h {
    bit<3>        version;                /** For GTPv1, this has a value of 1. */
    bit           protocol_type;          /** GTP (value 1) from GTP' (value 0) */
    bit           reserved;
    bit           extension_header_flag;  /** extension header optional field. */
    bit           seq_number_flag;        /** Sequence Number optional field */
    bit           n_pdu_number_flag;      /** N-PDU number optional field */
    bit<8>        message_type;           /** types of messages are defined in
3GPP TS 29.060 section 7.1 */
    bit<16>       message_length;         /** length of the payload in bytes */
    bit<32>       teid;                   /** Tunnel endpoint identifier */
    bit<16>       sequence_number;        /** optional */
    bit<8>        n_pdu_number;           /** optional */
    bit<8>        next_extension_hdr_type; /** optional if any of the E, S, or PN
bits are on. The field must be interpreted only if the E bit is on */
}

struct headers_t {
    NV_FIXED_HEADERS
    gtp_v1_h   gtpv1;
}

parser packet_parser(packet_in packet, out headers_t headers) {
    NV_FIXED_PARSER(packet, headers)

    @nv_transition_from("nv_parse_udp", GTP_U_PORT)
    state parse_gtp
```

```
    {
        packet.extract(headers.gtpv1);
        transition nv_parse_inner_ipv4;
    }
}

struct tunnel_headers_t {
    nv_ethernet_h ethernet;
    nv_ipv4_h     ipv4;
    nv_udp_h      udp;
    gtp_v1_h      gtpv1;
}

control c(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout nv_empty_metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    @nv_tunnel_fields(
        ethernet = {
            dst_addr = "variable",
            src_addr = "variable",
            ether_type = 0x0800
        },
        ipv4 = {
            version = 0x4,
            ihl = 0x5,
            diffserv = 0,
            ecn = "ignore",                 // cannot set
            total_len = "ignore",           // HW will calculate, cannot set
            identification = "ignore",      // cannot set
            flags = 0,
            frag_offset = 0,
            ttl = 64,
            protocol = 17,
```

```
        hdr_checksum = "ignore",      // HW will calculate, cannot set
        src_addr = "variable",
        dst_addr = "variable"
    },
    udp = {
        src_port = "ignore",          // HW will calculate entropy, cannot set
        dst_port = GTP_U_PORT,
        length   = "ignore",          // HW will calculate, cannot set
        checksum = "ignore"           // cannot set
    },
    gtpv1 = {
        version = 1,
        protocol_type = 1,
        reserved = 0,
        extension_header_flag = 1,
        seq_number_flag = 1,
        n_pdu_number_flag = 1,
        message_type = 0xFF,
        message_length = "variable",
        teid = "variable",
        sequence_number = 123,
        n_pdu_number = 255,
        next_extension_hdr_type = 0
    }
)
NvTunnelTemplate<tunnel_headers_t>() gtpv1Tunnel;

action drop() {
    nv_drop();
}

action forward(nv_logical_port_t port) {
    nv_send_to_port(port);
}

action tunnel(nv_ipv4_addr_t dst_addr, bit<32> teid) {
```

```
        nv_set_l3tunnel_underlay(headers, gtpv1Tunnel, {
            48w0x1, 48w0x2, 32w0x3, dst_addr,
            16w4, /* 4 bytes of optional GTP-U payload */
            teid
        });
    }

    table ip_as_key {
        key = {
            headers.ipv4.src_addr : exact;
        }
        actions = {
            NoAction;
            drop;
            forward;
            tunnel;
        }
        size = 128;
        default_action = NoAction;
        const entries = {
            (32w0x01010101) : tunnel(32w0x11111111, 1);
            (32w0x02020202) : tunnel(32w0x22222222, 2);
            (32w0x03030303) : tunnel(32w0x33333333, 3);
        }
    }

    table teid_as_key {
        key = {
            headers.gtpv1.teid : exact;
        }
        actions = {
            NoAction;
            drop;
            forward;
        }
        size = 128;
```

```
        default_action = forward(3);
        const entries = {
            (32w0x01) : forward(1);
            (32w0x02) : forward(2);

        }
    }

    apply {
        ip_as_key.apply();
        if (headers.gtpv1.isValid()) {
            teid_as_key.apply();
        }
        forward(3);
    }
}

NvDocaPipeline(
    packet_parser(),
    c()
) main;
```