Management Component Transport Protocol (MCTP)

net/mctp/ contains protocol support for MCTP, as defined by DMTF standard DSP0236. Physical interface drivers ("bindings" in the specification) are provided in drivers/net/mctp/.

The core code provides a socket-based interface to send and receive MCTP messages, through an AF_MCTP, SOCK_DGRAM socket.

Structure: interfaces & networks

The kernel models the local MCTP topology through two items: interfaces and networks.

An interface (or "link") is an instance of an MCTP physical transport binding (as defined by DSP0236, section 3.2.47), likely connected to a specific hardware device. This is represented as a `struct netdevice`.

A network defines a unique address space for MCTP endpoints by endpoint-ID (described by DSP0236, section 3.2.31). A network has a user-visible identifier to allow references from userspace. Route definitions are specific to one network.

Interfaces are associated with one network. A network may be associated with one or more interfaces.

If multiple networks are present, each may contain endpoint IDs (EIDs) that are also present on other networks.

Sockets API

Protocol definitions

MCTP uses `AF_MCTP` / `PF_MCTP` for the address- and protocol- families. Since MCTP is message-based, only `SOCK_DGRAM` sockets are supported.

```
int sd = socket(AF_MCTP, SOCK_DGRAM, 0);
```

The only (current) value for the `protocol` argument is 0.

As with all socket address families, source and destination addresses are specified with a `sockaddr` type, with a single-byte endpoint address:

```
typedef __u8                mctp_eid_t;

struct mctp_addr {
        mctp_eid_t          s_addr;
};

struct sockaddr_mctp {
        __kernel_sa_family_t smctp_family;
        unsigned int         smctp_network;
        struct mctp_addr     smctp_addr;
        __u8                 smctp_type;
        __u8                 smctp_tag;
};

#define MCTP_NET_ANY        0x0
#define MCTP_ADDR_ANY       0xff
```

Syscall behaviour

The following sections describe the MCTP-specific behaviours of the standard socket system calls. These behaviours have been chosen to map closely to the existing sockets APIs.

`bind()` : set local socket address

Sockets that receive incoming request packets will bind to a local address, using the `bind()` syscall.

```
struct sockaddr_mctp addr;

addr.smctp_family = AF_MCTP;
addr.smctp_network = MCTP_NET_ANY;
addr.smctp_addr.s_addr = MCTP_ADDR_ANY;
addr.smctp_type = MCTP_TYPE_PLDM;
addr.smctp_tag = MCTP_TAG_OWNER;

int rc = bind(sd, (struct sockaddr *)&addr, sizeof(addr));
```

This establishes the local address of the socket. Incoming MCTP messages that match the network, address, and message type will be received by this socket. The reference to 'incoming' is important here; a bound socket will only receive messages with the TO bit

set, to indicate an incoming request message, rather than a response.

The `smctp_tag` value will configure the tags accepted from the remote side of this socket. Given the above, the only valid value is `MCTP_TAG_OWNER`, which will result in remotely "owned" tags being routed to this socket. Since `MCTP_TAG_OWNER` is set, the 3 least-significant bits of `smctp_tag` are not used; callers must set them to zero.

A `smctp_network` value of `MCTP_NET_ANY` will configure the socket to receive incoming packets from any locally-connected network. A specific network value will cause the socket to only receive incoming messages from that network.

The `smctp_addr` field specifies a local address to bind to. A value of `MCTP_ADDR_ANY` configures the socket to receive messages addressed to any local destination EID.

The `smctp_type` field specifies which message types to receive. Only the lower 7 bits of the type is matched on incoming messages (ie., the most-significant IC bit is not part of the match). This results in the socket receiving packets with and without a message integrity check footer.

`sendto()`, `sendmsg()`, `send()` : transmit an MCTP message

An MCTP message is transmitted using one of the `sendto()`, `sendmsg()` or `send()` syscalls. Using `sendto()` as the primary example:

```
struct sockaddr_mctp addr;
char buf[14];
ssize_t len;

/* set message destination */
addr.smctp_family = AF_MCTP;
addr.smctp_network = 0;
addr.smctp_addr.s_addr = 8;
addr.smctp_tag = MCTP_TAG_OWNER;
addr.smctp_type = MCTP_TYPE_ECHO;

/* arbitrary message to send, with message-type header */
buf[0] = MCTP_TYPE_ECHO;
memcpy(buf + 1, "hello, world!", sizeof(buf) - 1);

len = sendto(sd, buf, sizeof(buf), 0,
                (struct sockaddr_mctp *)&addr, sizeof(addr));
```

The network and address fields of `addr` define the remote address to send to. If `smctp_tag` has the `MCTP_TAG_OWNER`, the kernel will ignore any bits set in `MCTP_TAG_VALUE`, and generate a tag value suitable for the destination EID. If `MCTP_TAG_OWNER` is not set, the message will be sent with the tag value as specified. If a tag value cannot be allocated, the system call will report an errno of `EAGAIN`.

The application must provide the message type byte as the first byte of the message buffer passed to `sendto()`. If a message integrity check is to be included in the transmitted message, it must also be provided in the message buffer, and the most-significant bit of the message type byte must be 1.

The `sendmsg()` system call allows a more compact argument interface, and the message buffer to be specified as a scatter-gather list. At present no ancillary message types (used for the `msg_control` data passed to `sendmsg()`) are defined.

Transmitting a message on an unconnected socket with `MCTP_TAG_OWNER` specified will cause an allocation of a tag, if no valid tag is already allocated for that destination. The (destination-eid,tag) tuple acts as an implicit local socket address, to allow the socket to receive responses to this outgoing message. If any previous allocation has been performed (to for a different remote EID), that allocation is lost.

Sockets will only receive responses to requests they have sent (with TO=1) and may only respond (with TO=0) to requests they have received.

`recvfrom()`, `recvmsg()`, `recv()` : receive an MCTP message

An MCTP message can be received by an application using one of the `recvfrom()`, `recvmsg()`, or `recv()` system calls. Using `recvfrom()` as the primary example:

```
struct sockaddr_mctp addr;
socklen_t addrlen;
char buf[14];
ssize_t len;

addrlen = sizeof(addr);

len = recvfrom(sd, buf, sizeof(buf), 0,
                (struct sockaddr_mctp *)&addr, &addrlen);

/* We can expect addr to describe an MCTP address */
assert(addrlen >= sizeof(buf));
assert(addr.smctp_family == AF_MCTP);
```

```
printf("received %zd bytes from remote EID %d\n", rc, addr.smctp_addr);
```

The address argument to `recvfrom` and `recvmsg` is populated with the remote address of the incoming message, including tag value (this will be needed in order to reply to the message).

The first byte of the message buffer will contain the message type byte. If an integrity check follows the message, it will be included in the received buffer.

The `recv()` system call behaves in a similar way, but does not provide a remote address to the application. Therefore, these are only useful if the remote address is already known, or the message does not require a reply.

Like the send calls, sockets will only receive responses to requests they have sent (TO=1) and may only respond (TO=0) to requests they have received.

`ioctl(SIOCMCTPALLOCTAG)` and `ioctl(SIOCMCTPDROPTAG)`

These tags give applications more control over MCTP message tags, by allocating (and dropping) tag values explicitly, rather than the kernel automatically allocating a per-message tag at `sendmsg()` time.

In general, you will only need to use these ioctls if your MCTP protocol does not fit the usual request/response model. For example, if you need to persist tags across multiple requests, or a request may generate more than one response. In these cases, the ioctls allow you to decouple the tag allocation (and release) from individual message send and receive operations.

Both ioctls are passed a pointer to a `struct mctp_ioc_tag_ctl`:

```
struct mctp_ioc_tag_ctl {
    mctp_eid_t      peer_addr;
    __u8            tag;
    __u16           flags;
};
```

`SIOCMCTPALLOCTAG` allocates a tag for a specific peer, which an application can use in future `sendmsg()` calls. The application populates the `peer_addr` member with the remote EID. Other fields must be zero.

On return, the `tag` member will be populated with the allocated tag value. The allocated tag will have the following tag bits set:

- `MCTP_TAG_OWNER`: it only makes sense to allocate tags if you're the tag owner
- `MCTP_TAG_PREALLOC`: to indicate to `sendmsg()` that this is a preallocated tag.
- ... and the actual tag value, within the least-significant three bits (`MCTP_TAG_MASK`). Note that zero is a valid tag value.

The tag value should be used as-is for the `smctp_tag` member of `struct sockaddr_mctp`.

`SIOCMCTPDROPTAG` releases a tag that has been previously allocated by a `SIOCMCTPALLOCTAG` ioctl. The `peer_addr` must be the same as used for the allocation, and the `tag` value must match exactly the tag returned from the allocation (including the `MCTP_TAG_OWNER` and `MCTP_TAG_PREALLOC` bits). The `flags` field must be zero.

Kernel internals

There are a few possible packet flows in the MCTP stack:

1. local TX to remote endpoint, message <= MTU:

```
sendmsg()
 -> mctp_local_output()
    : route lookup
    -> rt->output() (== mctp_route_output)
       -> dev_queue_xmit()
```

2. local TX to remote endpoint, message > MTU:

```
sendmsg()
-> mctp_local_output()
    -> mctp_do_fragment_route()
       : creates packet-sized skbs. For each new skb:
       -> rt->output() (== mctp_route_output)
          -> dev_queue_xmit()
```

3. remote TX to local endpoint, single-packet message:

```
mctp_pkttype_receive()
: route lookup
-> rt->output() (== mctp_route_input)
   : sk_key lookup
   -> sock_queue_rcv_skb()
```

4. remote TX to local endpoint, multiple-packet message:

```
mctp_pkttype_receive()
: route lookup
-> rt->output() (== mctp_route_input)
   : sk_key lookup
   : stores skb in struct sk_key->reasm_head

mctp_pkttype_receive()
: route lookup
-> rt->output() (== mctp_route_input)
   : sk_key lookup
   : finds existing reassembly in sk_key->reasm_head
   : appends new fragment
   -> sock_queue_rcv_skb()
```

Key refcounts

- keys are refed by:
  - a skb: during route output, stored in `skb->cb`.
  - netns and sock lists.
- keys can be associated with a device, in which case they hold a reference to the dev (set through `key->dev`, counted through `dev->key_count`). Multiple keys can reference the device.