J1939 Documentation

Overview / What Is J1939

SAE J1939 defines a higher layer protocol on CAN. It implements a more sophisticated addressing scheme and extends the maximum packet size above 8 bytes. Several derived specifications exist, which differ from the original J1939 on the application level, like MilCAN A, NMEA2000, and especially ISO-11783 (ISOBUS). This last one specifies the so-called ETP (Extended Transport Protocol), which has been included in this implementation. This results in a maximum packet size of ((2 ^ 24) - 1) * 7 bytes == 111 MiB.

Specifications used

- SAE J1939-21 : data link layer
- SAE J1939-81 : network management
- ISO 11783-6 : Virtual Terminal (Extended Transport Protocol)

Motivation

Given the fact there's something like SocketCAN with an API similar to BSD sockets, we found some reasons to justify a kernel implementation for the addressing and transport methods used by J1939.

- **Addressing:** when a process on an ECU communicates via J1939, it should not necessarily know its source address. Although, at least one process per ECU should know the source address. Other processes should be able to reuse that address. This way, address parameters for different processes cooperating for the same ECU, are not duplicated. This way of working is closely related to the UNIX concept, where programs do just one thing and do it well.
- **Dynamic addressing:** Address Claiming in J1939 is time critical. Furthermore, data transport should be handled properly during the address negotiation. Putting this functionality in the kernel eliminates it as a requirement for _every_ user space process that communicates via J1939. This results in a consistent J1939 bus with proper addressing.
- **Transport:** both TP & ETP reuse some PGNs to relay big packets over them. Different processes may thus use the same TP & ETP PGNs without actually knowing it. The individual TP & ETP sessions _must_ be serialized (synchronized) between different processes. The kernel solves this problem properly and eliminates the serialization (synchronization) as a requirement for _every_ user space process that communicates via J1939.

J1939 defines some other features (relaying, gateway, fast packet transport, ...). In-kernel code for these would not contribute to protocol stability. Therefore, these parts are left to user space.

The J1939 sockets operate on CAN network devices (see SocketCAN). Any J1939 user space library operating on CAN raw sockets will still operate properly. Since such a library does not communicate with the in-kernel implementation, care must be taken that these two do not interfere. In practice, this means they cannot share ECU addresses. A single ECU (or virtual ECU) address is used by the library exclusively, or by the in-kernel system exclusively.

J1939 concepts

PGN

The J1939 protocol uses the 29-bit CAN identifier with the following structure:

29 bit CAN-ID		
Bit positions within the CAN-ID		
28 ... 26	**25 ... 8**	**7 ... 0**
Priority	PGN	SA (Source Address)

The PGN (Parameter Group Number) is a number to identify a packet. The PGN is composed as follows:

PGN			
Bit positions within the CAN-ID			
25	**24**	**23 ... 16**	**15 ... 8**
R (Reserved)	DP (Data Page)	PF (PDU Format)	PS (PDU Specific)

In J1939-21 distinction is made between PDU1 format (where PF < 240) and PDU2 format (where PF >= 240). Furthermore, when using the PDU2 format, the PS-field contains a so-called Group Extension, which is part of the PGN. When using PDU2 format, the Group Extension is set in the PS-field.

PDU1 Format (specific) (peer to peer)	
Bit positions within the CAN-ID	
23 ... 16	**15 ... 8**
00h ... EFh	DA (Destination address)

PDU2 Format (global) (broadcast)	
Bit positions within the CAN-ID	
23 ... 16	**15 ... 8**
F0h ... FFh	GE (Group Extenstion)

On the other hand, when using PDU1 format, the PS-field contains a so-called Destination Address, which is _not_ part of the PGN. When communicating a PGN from user space to kernel (or vice versa) and PDU2 format is used, the PS-field of the PGN shall be set to zero. The Destination Address shall be set elsewhere.

Regarding PGN mapping to 29-bit CAN identifier, the Destination Address shall be get/set from/to the appropriate bits of the identifier by the kernel.

Addressing

Both static and dynamic addressing methods can be used.

For static addresses, no extra checks are made by the kernel and provided addresses are considered right. This responsibility is for the OEM or system integrator.

For dynamic addressing, so-called Address Claiming, extra support is foreseen in the kernel. In J1939 any ECU is known by its 64-bit NAME. At the moment of a successful address claim, the kernel keeps track of both NAME and source address being claimed. This serves as a base for filter schemes. By default, packets with a destination that is not locally will be rejected.

Mixed mode packets (from a static to a dynamic address or vice versa) are allowed. The BSD sockets define separate API calls for getting/setting the local & remote address and are applicable for J1939 sockets.

Filtering

J1939 defines white list filters per socket that a user can set in order to receive a subset of the J1939 traffic. Filtering can be based on:

- SA
- SOURCE_NAME
- PGN

When multiple filters are in place for a single socket, and a packet comes in that matches several of those filters, the packet is only received once for that socket.

How to Use J1939

API Calls

On CAN, you first need to open a socket for communicating over a CAN network. To use J1939, `#include <linux/can/j1939.h>`. From there, `<linux/can.h>` will be included too. To open a socket, use:

```
s = socket(PF_CAN, SOCK_DGRAM, CAN_J1939);
```

J1939 does use `SOCK_DGRAM` sockets. In the J1939 specification, connections are mentioned in the context of transport protocol sessions. These still deliver packets to the other end (using several CAN packets). `SOCK_STREAM` is not supported.

After the successful creation of the socket, you would normally use the `bind(2)` and/or `connect(2)` system call to bind the socket to a CAN interface. After binding and/or connecting the socket, you can `read(2)` and `write(2)` from/to the socket or use `send(2)`, `sendto(2)`, `sendmsg(2)` and the `recv*()` counterpart operations on the socket as usual. There are also J1939 specific socket options described below.

In order to send data, a `bind(2)` must have been successful. `bind(2)` assigns a local address to a socket.

Different from CAN is that the payload data is just the data that get sends, without its header info. The header info is derived from the sockaddr supplied to `bind(2)`, `connect(2)`, `sendto(2)` and `recvfrom(2)`. A `write(2)` with size 4 will result in a packet with 4 bytes.

The sockaddr structure has extensions for use with J1939 as specified below:

```
struct sockaddr_can {
   sa_family_t can_family;
   int         can_ifindex;
   union {
      struct {
         __u64 name;
```

```
                /* pgn:
                 * 8 bit: PS in PDU2 case, else 0
                 * 8 bit: PF
                 * 1 bit: DP
                 * 1 bit: reserved
                 */
        __u32 pgn;
        __u8  addr;
    } j1939;
  } can_addr;
}
```

`can_family` & `can_ifindex` serve the same purpose as for other SocketCAN sockets.

`can_addr.j1939.pgn` specifies the PGN (max 0x3ffff). Individual bits are specified above.

`can_addr.j1939.name` contains the 64-bit J1939 NAME.

`can_addr.j1939.addr` contains the address.

The `bind(2)` system call assigns the local address, i.e. the source address when sending packages. If a PGN during `bind(2)` is set, it's used as a RX filter. I.e. only packets with a matching PGN are received. If an ADDR or NAME is set it is used as a receive filter, too. It will match the destination NAME or ADDR of the incoming packet. The NAME filter will work only if appropriate Address Claiming for this name was done on the CAN bus and registered/cached by the kernel.

On the other hand `connect(2)` assigns the remote address, i.e. the destination address. The PGN from `connect(2)` is used as the default PGN when sending packets. If ADDR or NAME is set it will be used as the default destination ADDR or NAME. Further a set ADDR or NAME during `connect(2)` is used as a receive filter. It will match the source NAME or ADDR of the incoming packet.

Both `write(2)` and `send(2)` will send a packet with local address from `bind(2)` and the remote address from `connect(2)`. Use `sendto(2)` to overwrite the destination address.

If `can_addr.j1939.name` is set (!= 0) the NAME is looked up by the kernel and the corresponding ADDR is used. If `can_addr.j1939.name` is not set (== 0), `can_addr.j1939.addr` is used.

When creating a socket, reasonable defaults are set. Some options can be modified with `setsockopt(2)` & `getsockopt(2)`.

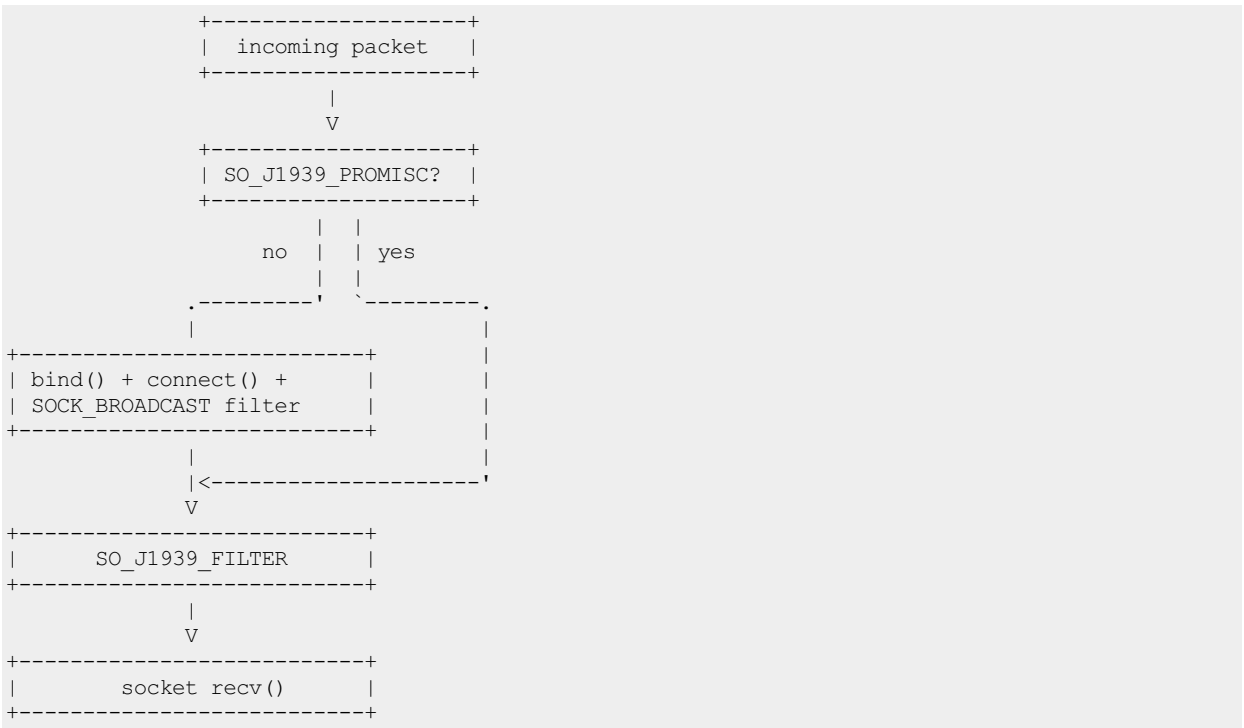
RX path related options:

- `SO_J1939_FILTER` - configure array of filters
- `SO_J1939_PROMISC` - disable filters set by `bind(2)` and `connect(2)`

By default no broadcast packets can be send or received. To enable sending or receiving broadcast packets use the socket option `SO_BROADCAST`:

```
int value = 1;
setsockopt(sock, SOL_SOCKET, SO_BROADCAST, &value, sizeof(value));
```

The following diagram illustrates the RX path:



TX path related options: `SO_J1939_SEND_PRIO` - change default send priority for the socket

Message Flags during send() and Related System Calls

`send(2)`, `sendto(2)` and `sendmsg(2)` take a 'flags' argument. Currently supported flags are:

- `MSG_DONTWAIT`, i.e. non-blocking operation.

recvmsg(2)

In most cases `recvmsg(2)` is needed if you want to extract more information than `recvfrom(2)` can provide. For example package priority and timestamp. The Destination Address, name and packet priority (if applicable) are attached to the msghdr in the `recvmsg(2)` call. They can be extracted using `cmsg(3)` macros, with `cmsg_level == SOL_J1939 && cmsg_type == SCM_J1939_DEST_ADDR`, `SCM_J1939_DEST_NAME` or `SCM_J1939_PRIO`. The returned data is a `uint8_t` for `priority` and `dst_addr`, and `uint64_t` for `dst_name`.

```
uint8_t priority, dst_addr;
uint64_t dst_name;

for (cmsg = CMSG_FIRSTHDR(&msg); cmsg; cmsg = CMSG_NXTHDR(&msg, cmsg)) {
        switch (cmsg->cmsg_level) {
        case SOL_CAN_J1939:
                if (cmsg->cmsg_type == SCM_J1939_DEST_ADDR)
                        dst_addr = *CMSG_DATA(cmsg);
                else if (cmsg->cmsg_type == SCM_J1939_DEST_NAME)
                        memcpy(&dst_name, CMSG_DATA(cmsg), cmsg->cmsg_len - CMSG_LEN(0));
                else if (cmsg->cmsg_type == SCM_J1939_PRIO)
                        priority = *CMSG_DATA(cmsg);
                break;
        }
}
```

Dynamic Addressing

Distinction has to be made between using the claimed address and doing an address claim. To use an already claimed address, one has to fill in the `j1939.name` member and provide it to `bind(2)`. If the name had claimed an address earlier, all further messages being sent will use that address. And the `j1939.addr` member will be ignored.

An exception on this is PGN 0x0ee00. This is the "Address Claim/Cannot Claim Address" message and the kernel will use the `j1939.addr` member for that PGN if necessary.

To claim an address following code example can be used:

```
struct sockaddr_can baddr = {
        .can_family = AF_CAN,
        .can_addr.j1939 = {
                .name = name,
                .addr = J1939_IDLE_ADDR,
                .pgn = J1939_NO_PGN,    /* to disable bind() rx filter for PGN */
        },
        .can_ifindex = if_nametoindex("can0"),
};

bind(sock, (struct sockaddr *)&baddr, sizeof(baddr));

/* for Address Claiming broadcast must be allowed */
int value = 1;
setsockopt(sock, SOL_SOCKET, SO_BROADCAST, &value, sizeof(value));

/* configured advanced RX filter with PGN needed for Address Claiming */
const struct j1939_filter filt[] = {
        {
                .pgn = J1939_PGN_ADDRESS_CLAIMED,
                .pgn_mask = J1939_PGN_PDU1_MAX,
        }, {
                .pgn = J1939_PGN_REQUEST,
                .pgn_mask = J1939_PGN_PDU1_MAX,
        }, {
                .pgn = J1939_PGN_ADDRESS_COMMANDED,
                .pgn_mask = J1939_PGN_MAX,
        },
};

setsockopt(sock, SOL_CAN_J1939, SO_J1939_FILTER, &filt, sizeof(filt));

uint64_t dat = htole64(name);
const struct sockaddr_can saddr = {
        .can_family = AF_CAN,
        .can_addr.j1939 = {
                .pgn = J1939_PGN_ADDRESS_CLAIMED,
                .addr = J1939_NO_ADDR,
        },
```

```
};

/* Afterwards do a sendto(2) with data set to the NAME (Little Endian). If the
 * NAME provided, does not match the j1939.name provided to bind(2), EPROTO
 * will be returned.
 */
sendto(sock, dat, sizeof(dat), 0, (const struct sockaddr *)&saddr, sizeof(saddr));
```

If no-one else contests the address claim within 250ms after transmission, the kernel marks the NAME-SA assignment as valid. The valid assignment will be kept among other valid NAME-SA assignments. From that point, any socket bound to the NAME can send packets.

If another ECU claims the address, the kernel will mark the NAME-SA expired. No socket bound to the NAME can send packets (other than address claims). To claim another address, some socket bound to NAME, must `bind(2)` again, but with only `j1939.addr` changed to the new SA, and must then send a valid address claim packet. This restarts the state machine in the kernel (and any other participant on the bus) for this NAME.

`can-utils` also include the `j1939acd` tool, so it can be used as code example or as default Address Claiming daemon.

Send Examples

Static Addressing

This example will send a PGN (0x12300) from SA 0x20 to DA 0x30.

Bind:

```
struct sockaddr_can baddr = {
        .can_family = AF_CAN,
        .can_addr.j1939 = {
                .name = J1939_NO_NAME,
                .addr = 0x20,
                .pgn = J1939_NO_PGN,
        },
        .can_ifindex = if_nametoindex("can0"),
};

bind(sock, (struct sockaddr *)&baddr, sizeof(baddr));
```

Now, the socket 'sock' is bound to the SA 0x20. Since no `connect(2)` was called, at this point we can use only `sendto(2)` or `sendmsg(2)`.

Send:

```
const struct sockaddr_can saddr = {
        .can_family = AF_CAN,
        .can_addr.j1939 = {
                .name = J1939_NO_NAME;
                .addr = 0x30,
                .pgn = 0x12300,
        },
};

sendto(sock, dat, sizeof(dat), 0, (const struct sockaddr *)&saddr, sizeof(saddr));
```