# The UCAN Protocol

UCAN is the protocol used by the microcontroller-based USB-CAN adapter that is integrated on System-on-Modules from Theobroma Systems and that is also available as a standalone USB stick.

The UCAN protocol has been designed to be hardware-independent. It is modeled closely after how Linux represents CAN devices internally. All multi-byte integers are encoded as Little Endian.

All structures mentioned in this document are defined in `drivers/net/can/usb/ucan.c`.

## USB Endpoints

UCAN devices use three USB endpoints:

CONTROL endpoint
: The driver sends device management commands on this endpoint

IN endpoint
: The device sends CAN data frames and CAN error frames

OUT endpoint
: The driver sends CAN data frames on the out endpoint

## CONTROL Messages

UCAN devices are configured using vendor requests on the control pipe.

To support multiple CAN interfaces in a single USB device all configuration commands target the corresponding interface in the USB descriptor.

The driver uses `ucan_ctrl_command_in/out` and `ucan_device_request_in` to deliver commands to the device.

### Setup Packet

| | |
|---|---|
| `bmRequestType` | Direction \| Vendor \| (Interface or Device) |
| `bRequest` | Command Number |
| `wValue` | Subcommand Number (16 Bit) or 0 if not used |
| `wIndex` | USB Interface Index (0 for device commands) |
| `wLength` | • Host to Device - Number of bytes to transmit<br>• Device to Host - Maximum Number of bytes to receive. If the device send less. Commom ZLP semantics are used. |

### Error Handling

The device indicates failed control commands by stalling the pipe.

### Device Commands

#### UCAN_DEVICE_GET_FW_STRING

*Dev2Host; optional*

Request the device firmware string.

### Interface Commands

#### UCAN_COMMAND_START

*Host2Dev; mandatory*

Bring the CAN interface up.

Payload Format
: `ucan_ctl_payload_t.cmd_start`

| | |
|---|---|
| mode | or mask of `UCAN_MODE_*` |

#### UCAN_COMMAND_STOP

*Host2Dev; mandatory*

Stop the CAN interface

Payload Format
*empty*

**UCAN_COMMAND_RESET**

*Host2Dev; mandatory*

Reset the CAN controller (including error counters)

Payload Format
*empty*

**UCAN_COMMAND_GET**

*Host2Dev; mandatory*

Get Information from the Device

**Subcommands**

UCAN_COMMAND_GET_INFO

Request the device information structure `ucan_ctl_payload_t.device_info`.

See the `device_info` field for details, and `uapi/linux/can/netlink.h` for an explanation of the `can_bittiming fields`.

Payload Format
`ucan_ctl_payload_t.device_info`

UCAN_COMMAND_GET_PROTOCOL_VERSION

Request the device protocol version `ucan_ctl_payload_t.protocol_version`. The current protocol version is 3.

Payload Format
`ucan_ctl_payload_t.protocol_version`

> **Note**
> Devices that do not implement this command use the old protocol version 1

**UCAN_COMMAND_SET_BITTIMING**

*Host2Dev; mandatory*

Setup bittiming by sending the structure `ucan_ctl_payload_t.cmd_set_bittiming` (see `struct bittiming` for details)

Payload Format
`ucan_ctl_payload_t.cmd_set_bittiming`.

**UCAN_SLEEP/WAKE**

*Host2Dev; optional*

Configure sleep and wake modes. Not yet supported by the driver.

**UCAN_FILTER**

*Host2Dev; optional*

Setup hardware CAN filters. Not yet supported by the driver.

### Allowed interface commands

| Legal Device State | Command | New Device State |
|---|---|---|
| stopped | SET_BITTIMING | stopped |
| stopped | START | started |
| started | STOP or RESET | stopped |
| stopped | STOP or RESET | stopped |
| started | RESTART | started |
| any | GET | *no change* |

## IN Message Format

A data packet on the USB IN endpoint contains one or more `ucan_message_in` values. If multiple messages are batched in a USB data packet, the `len` field can be used to jump to the next `ucan_message_in` value (take care to sanity-check the `len` value against

the actual data size).

### `len` field

Each `ucan_message_in` must be aligned to a 4-byte boundary (relative to the start of the start of the data buffer). That means that there may be padding bytes between multiple `ucan_message_in` values:

```
+----------------------------+ < 0
|                            |
|   struct ucan_message_in   |
|                            |
+----------------------------+ < len
          [padding]
+----------------------------+ < round_up(len, 4)
|                            |
|   struct ucan_message_in   |
|                            |
+----------------------------+
            [...]
```

### `type` field

The `type` field specifies the type of the message.

#### UCAN_IN_RX

`subtype`
: zero

Data received from the CAN bus (ID + payload).

#### UCAN_IN_TX_COMPLETE

`subtype`
: zero

The CAN device has sent a message to the CAN bus. It answers with a list of tuples <echo-ids, flags>.

The echo-id identifies the frame from (echos the id from a previous UCAN_OUT_TX message). The flag indicates the result of the transmission. Whereas a set Bit 0 indicates success. All other bits are reserved and set to zero.

### Flow Control

When receiving CAN messages there is no flow control on the USB buffer. The driver has to handle inbound message quickly enough to avoid drops. I case the device buffer overflow the condition is reported by sending corresponding error frames (see :ref:`can_ucan_error_handling`)

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\networking\[linux-master][Documentation][networking]can_ucan_protocol.rst, line 245); *backlink***
>
> Unknown interpreted text role "ref".

## OUT Message Format

A data packet on the USB OUT endpoint contains one or more `struct ucan_message_out` values. If multiple messages are batched into one data packet, the device uses the `len` field to jump to the next ucan_message_out value. Each ucan_message_out must be aligned to 4 bytes (relative to the start of the data buffer). The mechanism is same as described in :ref:`can_ucan_in_message_len`.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\networking\[linux-master][Documentation][networking]can_ucan_protocol.rst, line 255); *backlink***
>
> Unknown interpreted text role "ref".

```
+----------------------------+ < 0
|                            |
|   struct ucan_message_out  |
|                            |
+----------------------------+ < len
          [padding]
+----------------------------+ < round_up(len, 4)
```

```
|                            |
|   struct ucan_message_out  |
|                            |
+----------------------------+
            [...]
```

### `type` field

In protocol version 3 only `UCAN_OUT_TX` is defined, others are used only by legacy devices (protocol version 1).

#### UCAN_OUT_TX

`subtype`
: echo id to be replied within a CAN_IN_TX_COMPLETE message

Transmit a CAN frame. (parameters: `id`, `data`)

### Flow Control

When the device outbound buffers are full it starts sending *NAKs* on the *OUT* pipe until more buffers are available. The driver stops the queue when a certain threshold of out packets are incomplete.

## CAN Error Handling

If error reporting is turned on the device encodes errors into CAN error frames (see `uapi/linux/can/error.h`) and sends it using the IN endpoint. The driver updates its error statistics and forwards it.

Although UCAN devices can suppress error frames completely, in Linux the driver is always interested. Hence, the device is always started with the `UCAN_MODE_BERR_REPORT` set. Filtering those messages for the user space is done by the driver.

### Bus OFF

- The device does not recover from bus of automatically.
- Bus OFF is indicated by an error frame (see `uapi/linux/can/error.h`)
- Bus OFF recovery is started by `UCAN_COMMAND_RESTART`
- Once Bus OFF recover is completed the device sends an error frame indicating that it is on ERROR-ACTIVE state.
- During Bus OFF no frames are sent by the device.
- During Bus OFF transmission requests from the host are completed immediately with the success bit left unset.

## Example Conversation

1. Device is connected to USB
2. Host sends command `UCAN_COMMAND_RESET`, subcmd 0
3. Host sends command `UCAN_COMMAND_GET`, subcmd `UCAN_COMMAND_GET_INFO`
4. Device sends `UCAN_IN_DEVICE_INFO`
5. Host sends command `UCAN_OUT_SET_BITTIMING`
6. Host sends command `UCAN_COMMAND_START`, subcmd 0, mode `UCAN_MODE_BERR_REPORT`