Volvo DiCE USB Protocol

A hardware-verified implementation guide

Status: draft 0.2, 2026-08-31

This document describes the verified subset of the native USB protocol used by Volvo DiCE. It is intended to let another developer implement DiCE transport support without depending on the Python `linux_dice` reference implementation.

This is an independent interoperability document. It is not a complete DiCE specification, a Windows J2534 DLL, or a description of Volvo diagnostic services.

Revision history

Draft	Change
`0.1`	Initial protocol extraction from the working implementation
`0.2`	Byte-level audit, evidence corrections, source traceability, and implementation checklist

1. Evidence labels

Statements use these confidence labels:

- **HW** — exercised successfully against a physical DiCE on Linux.
- **DLL** — reconstructed from `TSDiCE32.dll` behavior or data layout.
- **OBS** — decoded from an observed DiCE message or capture.
- **OPEN** — not yet established; confirm with VIDA USB captures.

Verified hardware consisted of a Volvo DiCE and a powered 2011 XC90 P2 CEM. LS-CAN receive and one bounded request/response operation were proven. HS-CAN controller configuration was accepted, but no live HS traffic was present on that bench. Simultaneous HS/LS operation has not yet been verified.

2. USB transport

Property	Value	Evidence
Vendor ID	`0x17AA`	HW
Product ID	`0xD1CE`	HW
Interface	`0`	HW
Bulk OUT endpoint	`0x01`	HW
Bulk IN endpoint	`0x81`	HW

Select the device configuration, claim interface 0, write framed requests to endpoint `0x01`, and continuously read endpoint `0x81`. The reference implementation requests up to 4096 bytes per read; this is an implementation choice, not a discovered protocol limit. One observed bulk read contained two complete B5 frames. A defensive implementation must also buffer a partial B5 frame across reads because USB bulk-transfer boundaries are not application message boundaries. **HW/OBS**

3. B5 envelope

Every request, reply, and unsolicited indication is carried in this envelope:

```
B5 LL HH HC [payload...] PC
```

Field	Meaning
`B5`	Frame marker
`LL HH`	Little-endian body length: payload length plus one checksum byte
`HC`	`(0x69 + B5 + LL + HH) & 0xFF`
`payload`	One DiCE command, reply, or indication
`PC`	`(0x69 + sum(payload)) & 0xFF`

The complete envelope length is `body_length + 4`. Reject a bad marker, zero body length, incorrect declared length, or either checksum mismatch. Hardware exchanges confirmed the envelope and checksums. Handling concatenated frames was required by an observed transfer; buffering an incomplete frame is the corresponding defensive stream-decoder requirement. **HW/OBS**

Example request carrying payload `62`:

```
B5 02 00 20 62 CB
```

4. Message model

The first payload byte identifies the operation. Ordinary replies observed so far repeat the request command in byte zero. CAN traffic arrives asynchronously as command `0x81`. **HW/OBS**

An implementation should have one reader for bulk IN and dispatch complete B5 payloads into:

1. replies awaited by a control request; and
2. unsolicited `0x81` CAN indications.

Do not assume the next USB read is necessarily the reply to the most recent

write when CAN controllers are running.

5. Verified device-information commands

These operations do not require a CAN controller.

Command	Request payload	Successful reply payload	Meaning	Evidence
`0x00`	`00`	`00 VV VV VV VV`	RFC version bytes	HW
`0x02`	`02`	`02 VV`	Hardware version	HW
`0x03`	`03`	`03 VV VV VV VV`	Firmware version read	HW
`0x5F`	`5F`	`5F 00 YY YY MM DD`	Warranty date; year is LE	HW
`0x62`	`62`	`62 00 VV VV`	Supply millivolts, unsigned LE	HW

Observed examples:

```
TX payload  00
RX payload  00 00 01 01 18

TX payload  02
RX payload  02 01

TX payload  03
RX payload  03 00 01 04 03

TX payload  5F
RX payload  5F 00 D8 07 08 08

TX payload  62
RX payload  62 00 E6 31
```

For `0x5F` and `0x62`, reply byte 1 is a status byte and was zero on success.

6. Raw CAN lifecycle

The controller lifecycle accepted by the physical DiCE is:

```
allocate -> configure -> start -> add filters -> read/write
         -> remove filters -> stop -> free
```

All byte sequences below are B5 payloads; apply the envelope from section 3

before writing them to USB.

6.1 Allocate controller — `0x10`

```
Request:  10 CC
Reply:    10 SS HH
```

Field	Meaning
`CC`	Channel: `00` LS-CAN, `01` HS-CAN
SS	Status; zero was accepted as success
`HH`	Controller handle used by subsequent commands

HW

6.2 Set controller value — `0x14`

```
Request:  14 HH KK VV VV VV VV
Reply:    14 SS
```

KK is the setting key. The value is an unsigned 32-bit little-endian integer.

The following complete sequences are accepted by DiCE and are used by the working implementation. **HW/DLL**

Key	Operational meaning	LS value	HS value
`0`	CAN/controller type	`2`	`2`
`1`	Bit rate	`125000`	`500000`
`3`	Bus mode	`0` silent, `1` active	`0` silent, `1` active
`4`	Timing/sample value A	`68`	`80`
`5`	Timing/sample value B	`35`	`20`
`7`	Channel-specific final flag	`1`	`0`

Only key `1` (bit rate) and key `3` (silent versus active operation) have names supported by their observed effect and DLL mapping. The names for keys `0`, `4`, `5`, and `7` are intentionally descriptive placeholders. Preserve the complete value sets unless stronger evidence establishes their formal meaning or a general timing formula. **OPEN**

6.3 Start controller — `0x12`

```
Request:  12 HH
Reply:    12 SS
```

Zero status indicates success. **HW**

6.4 Stop controller — `0x13`

```
Request:  13 HH
Reply:    13 SS
```

Remove installed filters before stopping. **HW**

6.5 Free controller — `0x11`

```
Request:  11 HH
Reply:    11 SS
```

Free the handle even when earlier cleanup fails. **HW**

7. Receive filters

7.1 Add filter — 0x1A

```
Request:  1A HH FF FF 01 [12-byte mask] [12-byte pattern]
Reply:    1A SS II
```

Field	Meaning
FF FF	Little-endian flags: 00 00 for 11-bit, 01 00 for 29-bit
01	Filter type used for the verified all-pass filter
mask	Twelve zero bytes for all-pass
pattern	Twelve zero bytes for all-pass
II	Returned filter identifier

The flag is numerically 0x0001 for 29-bit identifiers; its byte encoding is 01 00. Install both an 11-bit and a 29-bit all-pass filter when both formats must be received. **HW/DLL**

7.2 Remove filter — 0x1B

```
Request:  1B HH II
Reply:    1B SS
```

Remove filters in reverse installation order before stopping and freeing the controller. **HW**

Mask and pattern semantics beyond the verified all-zero all-pass case remain to be confirmed. **OPEN**

8. Raw CAN receive indication — 0x81

One observed classic-CAN receive payload is exactly 24 bytes:

```
Offset  Size  Meaning
0       1     81, unsolicited indication command
1       1     Controller handle
2       1     16, record length marker (0x16)
3       1     00, observed outer flags/reserved
4       2     Record flags, little-endian
6       6     DiCE timestamp, little-endian
12      4     CAN identifier, little-endian
16      8     CAN data
```

Record-flags bit 0 selects a 29-bit identifier. A zero value selects 11-bit.

Do not infer identifier format from the numeric identifier value. **DLL/OBS/HW**

Observed extended frame:

```
81 00 16 00 01 00 B6 B4 7F BC B0 02 FC 7F 21 00 01 4B 00 10 F0 00 00 00
```

Decoded:

```
handle       00
extended     yes
identifier   0x00217FFC
data         01 4B 00 10 F0 00 00 00
```

The six-byte timestamp unit, epoch, and rollover behavior are not established. The observed record always contains eight data bytes; the meaning of byte 2 and the encoding of shorter received CAN frames have not yet been independently proven. **OPEN**

9. Raw CAN transmit stream — `0x80`

9.1 Envelope

```
80 HH NN NN [16-byte record ...]
```

`NN NN` is the little-endian byte count of the record area, not the record count. Each classic-CAN transmit record is exactly 16 bytes. Multiple-record batches are supported by the DLL layout but have not been exercised by the Linux implementation. **DLL; single record HW**

9.2 Sixteen-byte transmit record

```
Offset  Size  Meaning
0       1     Record length: CAN data length + 8
1       1     Flags; bit 0 selects a 29-bit identifier
2       2     Transaction tag, little-endian
4       4     CAN identifier, little-endian
8       8     CAN data, zero-padded by the reference implementation
```

For an eight-byte extended frame with ID `0x000FFFFE`, tag zero, and data `CB 40 B9 FB 00 00 00 00`, the record is:

```
10 01 00 00 FE FF 0F 00 CB 40 B9 FB 00 00 00 00
```

With symbolic handle `HH`, the complete payload is:

```
80 HH 10 00 10 01 00 00 FE FF 0F 00 CB 40 B9 FB 00 00 00 00
```

The record-length byte is essential. A malformed experimental record beginning with `01 00 ...` caused the DiCE firmware to reset and disconnect. **DLL/OBS**

10. Transmit confirmation — `0x81`

The verified transmit-confirmation payload is 16 bytes:

```
Offset  Size  Meaning
0       1     81, unsolicited indication command
1       1     Controller handle
2       2     0E 00, confirmation record marker
4       4     Descriptor, little-endian
8       4     DiCE timestamp, little-endian
12      4     CAN identifier, little-endian
```

Observed example:

```
81 00 0E 00 09 00 00 00 7E AE 01 00 FE FF 0F 00
```

Decoded descriptor: `0x00000009`; identifier: `0x000FFFFE`. **HW/OBS**

This confirms that the DiCE firmware accepted the USB transmit record. It does not by itself prove electrical transmission, CAN acknowledgement, or an ECU response. Establish those separately from bus observation or matching received frames.

The meaning of descriptor bits other than the observed value `9`, and the relationship between transaction tags and confirmations, remain open. **OPEN**

11. Hardware-proven end-to-end example

The external validation program performed one read-only operation on LS-CAN:

```
CAN TX ID       0x000FFFFE, 29-bit
CAN TX data     CB 40 B9 FB 00 00 00 00
DiCE confirm    descriptor 0x00000009
CAN RX ID       0x00800003, 29-bit
Result          known bench VIN YV4952CY5B1606975
```

This proves USB framing, LS controller setup, filters, one transmit record, confirmation decoding, receive decoding, and cleanup as one path. It does not make the diagnostic request or VIN decoder part of the transport protocol. **HW**

The `0x0003` response identifier reported by older Volvo diagnostic prior art is the low 16 bits of the full DiCE identifier `0x00800003` observed here.

12. Recommended implementation behavior

- Use one buffered B5 stream decoder per DiCE connection.
- Validate both checksums and exact message shapes before decoding fields.
- Route unsolicited `0x81` indications independently of command replies.
- Treat controller and filter identifiers as opaque values returned by DiCE.
- Use bounded timeouts based on an absolute monotonic deadline.
- Preserve received CAN frames encountered while awaiting a TX confirmation.
- Require explicit application opt-in before enabling active transmission.
- On failure, attempt filter removal, controller stop, controller free, and USB interface release without allowing one cleanup error to skip later steps.
- Do not treat a TX confirmation as an ECU response.

Minimal implementation checklist

- [ ] Discover `17AA:D1CE`, select its configuration, and claim interface 0.
- [ ] Implement a buffered B5 encoder/decoder with both checksum validations.
- [ ] Separate awaited command replies from unsolicited `0x81` indications.
- [ ] Allocate one controller and retain the returned handle.
- [ ] Apply all six channel settings in the documented order.
- [ ] Start the controller and retain every returned filter identifier.
- [ ] Decode receive flags, timestamp, identifier, and data using the documented little-endian fields.
- [ ] For transmission, emit a 16-byte record with a valid length byte and wait for a matching confirmation identifier under an absolute deadline.
- [ ] Queue receive indications that arrive while waiting for confirmation.
- [ ] Remove filters, stop, free, release the interface, and preserve the first cleanup error without skipping later cleanup operations.

13. J2534 concept mapping

J2534 concept	Native DiCE operation
`PassThruOpen`	Open USB `17AA:D1CE`, claim interface 0
`PassThruConnect(CAN, ...)`	Allocate `0x10`, configure `0x14`, start `0x12`
`PassThruStartMsgFilter`	Add filter `0x1A`
`PassThruWriteMsgs`	Transmit stream `0x80`
`PassThruReadMsgs`	Receive indication `0x81`
`PassThruStopMsgFilter`	Remove filter `0x1B`
`PassThruDisconnect`	Stop `0x13`, free `0x11`
`PassThruClose`	Release USB interface

This mapping is conceptual. The documented protocol is not a binary-compatible J2534 DLL, and ISO 15765 transport is not implemented by the verified subset.

14. Open questions for VIDA USB capture

A VIDA/DiCE capture on a live car would be most valuable for:

1. Simultaneous HS- and LS-CAN controller allocation and traffic.
2. Exact initialization order and any commands omitted by the minimal Linux implementation.
3. Non-all-pass filter mask and pattern behavior.

4. Short-DLC receive records and other `0x81` indication variants.
5. Transaction tags, descriptor values, error confirmations, and batching.
6. Periodic messages, flow control, and ISO 15765 responsibilities.
7. Recovery behavior after USB timeout, unplug, or vehicle power transition.

Capture raw USB traffic from before VIDA opens DiCE until after VIDA closes it. Record the VIDA action and vehicle state, and keep programming or configuration operations out of the first capture.

15. Evidence traceability

The following repository artifacts support this draft:

Subject	Primary evidence
B5 envelope and stream handling	`linux_dice/framing.py`, `test_linux_dice.py`
USB identity and endpoints	`linux_dice/transport.py`, physical USB operation
Device-information exchanges	`DICE_READONLY_PROTOCOL.md`, `linux_dice/device.py`
CAN lifecycle and filters	`linux_dice/passive_can.py`, hardware session results
Receive record	observed frame in `test_linux_dice.py`, DLL flag mapping
Transmit record	`resource-db/DICE_CAN_TX_RECORD.md`, DLL conversion routine
Confirmation record	observed frame in `test_linux_dice.py`, hardware confirmation
End-to-end LS result	`dice_cem_vin.py`, verified bench run

Offline tests are regression evidence for byte layout and error handling. They do not independently elevate a DLL-derived or inferred field to hardware proof.

16. Reference implementation and license

The byte-level reference implementation is the `linux_dice` package in this repository. Its core files are:

- `linux_dice/framing.py`
- `linux_dice/transport.py`
- `linux_dice/passive_can.py`
- `linux_dice/active_can.py`

The project is licensed under GNU LGPL version 3. See `COPYING.LESSER` and `COPYING`.