25-09-25

Technical description for:

BE Boot Loader

**Purpose:**

This technical document brings the implementation details for the BE boot loader, down to:

* Process sequences
* Detailed description of messages in use
* Detailed addresses and contents.
* GUI contents

At the end of this guide are:

* Appendix that brings useful manual excerpts
* TBD management.

Contents

[SAE J1939 protocol 3](#_Toc209330262)

[General 29bit address structure 4](#_Toc209330263)

[Claim address for the downloader 4](#_Toc209330264)

[Enter boot mode 6](#_Toc209330265)

[Working procedure 8](#_Toc209330266)

[The user interface 8](#_Toc209330267)

[Preparation to load 9](#_Toc209330268)

[The download procedure 10](#_Toc209330269)

[F28069 specifics 11](#_Toc209330270)

[Appendix: Manual excerpts 13](#_Toc209330271)

[Boot behavior 13](#_Toc209330272)

[TBD management 14](#_Toc209330273)

# SAE J1939 protocol

**Motivation**

The unit for firmware downloading is an ECU in a vehicle.

The vehicle complies with the SAEJ1939 (in the sequel referred simply as J1939) CAN protocol.

The initial negotiation (explained below) prior to the boot process is made by the J1939 protocol.

The downloading process itself is done by another, completely non-standard protocol. Using a non-standard protocol is intentional, as obstacle for malicious code insertion.

Only a very small subset of the J1939 protocol is used:

* Address claim
* DM14 / DM15 enter boot protocol

Implementing a full J1939 stack in the boot sector does not make sense, for the limited flash resources there. Only the required messages are implemented, on a dumb “this is the byte sequence” basis.

The messages documented below are therefore detailed to the byte level.

**Source for learning:**

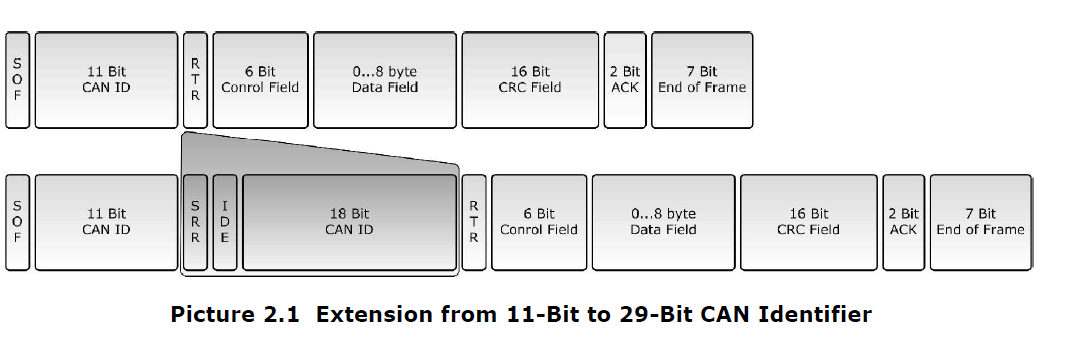
A Comprehensible Guide to J1939 by Wilfried Voss, Copperhill Media Corporation, ISBN: 978-0-9765116-3-2

**Abbreviations:**

|  |  |
| --- | --- |
| Abbreviation | For |
| DLC | Data Length in of message in bytes |
| DM | Diagnostic message |
| DP | Data Page bit, for us always zero. |
| ECU | Electronic Control Unit (a node in J1939 bus) |
| PDU | Process data unit |
| PGN | Parameter Group number |
| PF | PDU format (part of PGN) |
| PS | PDU specific (part of PGN) |
| R | Reserved |
| SA | Source address |
| SAE | Society of Automotive Engineering |

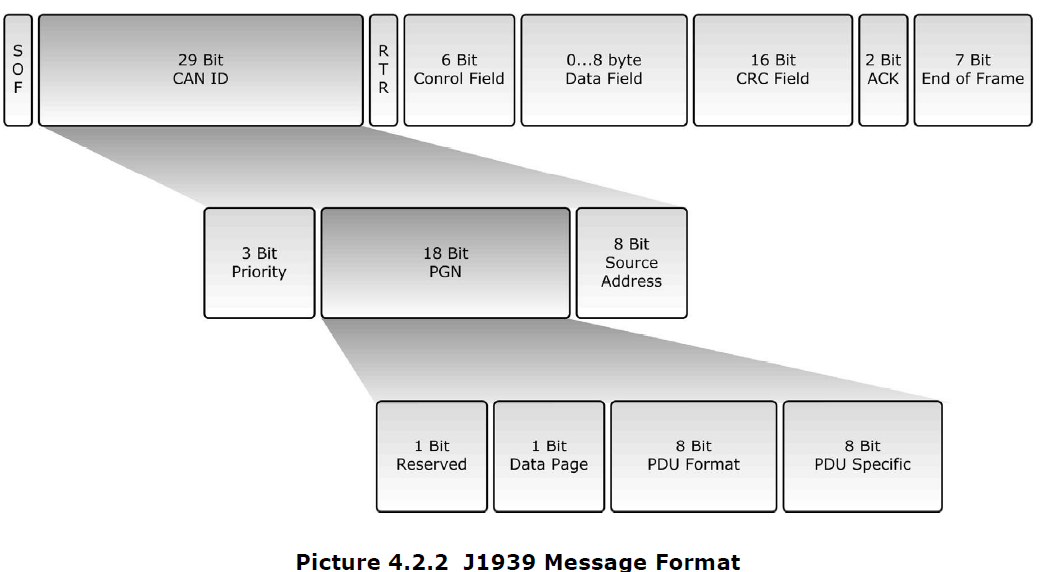
## General 29bit address structure

The ID of a CAN message can be 11bit or 29bit.



A message with CAN ID of 11 bit will normally arbitrate in priority to a 29-bit message, as in 11- bit message the IDE bit is dominant.

The J1939 message uses always 29bit ID, as depicted below.



## Claim address for the downloader

The claim is

Priority (3) = 110b

R (1) = 0

DP (1) = 0

PF (8) = 0xEA

PS (8) = xx = Requested address, 0x80 or what ever

SA (8) = 0x21

The 29bit preamble is 0x18EAxx21

The data field need be 0, 0xEE, 0 , 0xff , 0xff , 0xff , 0xff , 0xff

If this address is already claimed, we should get a response with the ID 0x18EE00xx and some contents. If the address is already claimed we go for the next address.

The extra contents describe as follows (contents are not really interesting for us):

| **Byte** | **Field** | **Size** | **Notes** |
| --- | --- | --- | --- |

|  |  |  |  |
| --- | --- | --- | --- |
| 1–3 | Identity Number | 21 b | Manufacturer-unique number |

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | ECU Instance | 3 b | Usually 0 |

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | Function Instance | 5 b |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 5 | Function | 8 b | Function code (e.g. “Engine” = 0) |

|  |  |  |  |
| --- | --- | --- | --- |
| 6 | Reserved / Vehicle Sys | 7 b | Reserved + system code |

|  |  |  |  |
| --- | --- | --- | --- |
| 6 | Arbitrary Address Capable | 1 b | 0 = No, 1 = Yes |

|  |  |  |  |
| --- | --- | --- | --- |
| 7 | Industry Group | 3 b | (0 = On-Highway, 1 = Agricultural, etc.) |

|  |  |  |  |
| --- | --- | --- | --- |
| 7 | Vehicle System Instance | 4 b |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 8 | Manufacturer Code | 11 b | SAE-assigned |

## Enter boot mode

The boot loader only deals with DM14/DM15 (enter boot mode/acknowledge boot mode)

The DM14 message is destined to a specific slave. Slave ID may be either default (TBD) if awakened in boot mode.

When the slave ID is claimed, the claimed slave ID shall be stored in the following struct residing in address 0x11f00. The struct is

struct BootInfo

{

long unsigned password 0x1234568 ;

short unsigned DefaultDeviceAddress ;

short unsigned ClaimedDeviceAddress ; // Equal to the default if none was claimed

}

The DM14 message (priority = 6 , data page 0, PF = 0xD9, PS = ECU address, SA = address of loader (manually selectable) data = 0,7,0,1,1,0xff,0xff,0xff, refer the data field definition for this PGN below.

| **Byte** | **Field** | **Notes** |
| --- | --- | --- |
| 1 | Command | Always **0x00** for “Initiate Download.” Other values are defined for different commands. |
| 2 | Max Number of Bytes per Segment (LSB) | Used with transport protocol (when firmware is segmented). LSB. |
| 3 | Max Number of Bytes per Segment (MSB) | MSB. |
| 4 | Number of Segments (LSB) | For some loaders, total number of segments expected. |
| 5 | Number of Segments (MSB) |  |
| 6 | Reserved / Control | Typically reserved = 0xFF. |
| 7 | Reserved | Typically 0xFF. |
| 8 | Reserved | Typically 0xFF. |

The DM15 message (priority = 6 , data page 0, PF = 0xD8, PS = sender address, SA = ECU address

Data is 0,0,0xff,0xff,0xff,0xff,0xff,0xff

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Byte** | **Field** | **Meaning** | | --- | --- | --- | | 1 | **Response Code** | Status of the request: • 0 = Acknowledge (positive) • 1 = NAK • 2 = Cannot respond right now | | 2 | **Group Function Value** | Echo of the command in DM14 (for download initiate, = 0). | | 3–4 | **Reserved** | Typically set to 0xFF. | | 5–8 | **Reserved** | Typically set to 0xFF. | |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

## Working procedure

### The user interface

The PC user interface described here is not final in the graphic design sense, but it presents the available controls.

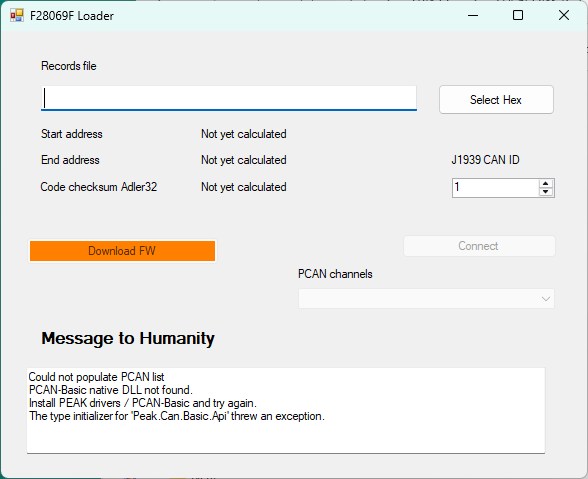


Figure : The GUI contents

|  |  |  |
| --- | --- | --- |
| Item | Description | Comment |
| Records file Edit box + Select Hex button | Allow selection of valid load file | The file is tested for sanity (format, addresses range) |
| Start address  End address | Actual code starts and end addresses. | Calculated on the basis of the actual contents of the hex file |
| Checksum Adler32 | Actual code checksum. | Calculated on the basis of the actual contents of the hex file |
| Download firmware button | Start the downloading process. | Enabled only after hex file is loaded and PCAN device is identified |
| PCAN channel combo + connect button | Scan and present all the present PCAN units.  Press connects to claim the PCAN device | Connection button only claims a device. It issues no actual communication |
| J1939 CAN ID | The ID of the ECU to be programmed | Must be known in advance, but the owner of that address shall be verified by its J1939 name. |
| Message to Humanity | Messages for the user | In the above figure it complains to PCAN device found. |

### Preparation to load

The following steps hold (the same steps are taken on the loader’s part regardless whether the target is in the boot state or operational.

|  |  |  |
| --- | --- | --- |
| # | Step | Comments |
| 1 | User sets the J1939 ECU address to download, and selects the correct image file. | Loader program uploads the image. Image is scanned for sanity. If failed downloading may not proceed. |
| 2 | Loader program claims address 0x80.  If it fails it claims 0x81 … till 0xFD, stepping automatically over if encountering the download target. | If none of these ECU addresses are free, claim fails and we can’t proceed. |
| 3 | Loader program claims target address.  Target is recognized by its name | If claim is not responded by a correct name, downloading will not proceed. |
| 4 | Loader program emits DM14, requesting download FW | If power is on, DM15 will respond “Cannot respond right now”.  If not responded by DM15 and with ACK, downloading will not proceed.  If this was a loaded application, it will fill the struct BootInfo defined above in address 0x11f00, and reset the device by arming the watchdog and entering an infinite SW loop. |
| 5 | Boot wakes up | Scan for well loaded application (Adler32 checksum ok, verse ok), if failed, stay in boot mode.  If ok, look for BootInfo. If valid, assume ID, and remain in boot mode. |
| 6 | Firmware download | The J1939 ID serves as 11 bit CAN ID for the remaining of the DOWNLOAD service. The 11-bit ID download process is described below. |

### The download procedure

The downloading procedure goes as follows.

|  |  |  |
| --- | --- | --- |
| # | Step | Comments |
| 1 | Clear flash and BootInfo | All the sectors B to H are erased, regardless of the contents to burn  Clear the BootInfo area. |
| 2 | Work sector H first down to sector B | If the sector contains no programmed addresses, it is ignored.  Otherwise, full 16K of data is put into the target, unconfirmed.  First a message is sent with the leading 3 bits as zero, and 8 bytes of contents as:   * First byte is the next sector to program * Bytes 1: 0 * Byte 2 – 3 : Sender address * Bytes 4..7 are 0x12345678   The message is responded with same contents, with the preamble of master ID.  Next each message has its leading 3 bits as sector selector, range 1(B) to 7(H), and 8 bits of slave address.  The data contents are   * 2 bytes start address * 6 bytes of data   Making total of 1093 messages (bytes exceeding the sector address at the last message are ignored)  With actual 5msec per message, this stage takes about 60 seconds per sector.  Next a message is sent with the leading 3 bits as zero, and 8 bytes of contents as:   * First two bytes are starting relative address * Next two bytes are burning length * Last 4 bytes are Adler32 checksum of entire sector.   Following check that all addresses are covered and that checksum fit, entire flash sector is programmed.  Answered with two bytes:  First is 1 for OK 0 for error  Second is error code, applicable on error.  Note that the statistics sector burns as 0xff, so its burning does nothing. |
| 3 | Burn statistics | With sector B still in memory, send content commands to populate the statistics region.  Send a burn command and receive acknowledge |
| 4 | Reboot | Resend the DM14 sequence. With no BootInfo and well-burnt code, the DSP shall start normally. |

# F28069 specifics

**Memory Map:**

The flash memory of 28069F is divided as follows:

|  |  |  |
| --- | --- | --- |
| Start address | Length | Name |
| 0x3D8000 | 0x4000 = 16K | H |
| 0x3DC000 | 0x4000 = 16K | G |
| 0x3E0000 | 0x4000 = 16K | F |
| 0x3E4000 | 0x4000 = 16K | E |
| 0x3E8000 | 0x4000 = 16K | D |
| 0x3EC000 | 0x4000 = 16K | C |
| 0x3F0000 | 0x4000 = 16K | B |
| 0x3F4000 | 0x3FF8 = 16K-8 word | A |

**Boot sector**

The start address of the code is 0x3F 7FF6. (See the section on Boot Behavior below)

The boot sector must be Sector A unless we want to program the OTP, a risky business as an error can kill the DSP.

This means that the flash password cannot be dealt with without deleting the boot, and we assume that the flash remains unlocked all the time with the addresses 0x3F 7FF8 …. 0x3F 7FFF unprogrammed (reads 0xffffffffffffffff).

**Valid program range**

The least valid program address is 0x3D8000

The maximum valid program address is 0x3F3EFF

**Statistics range**

The statistics range is 0x3F3F00 to 0x3F3FFF.

The contents of this address range are:

|  |  |  |
| --- | --- | --- |
| Address | Length | Contents |
| 0x3F3F00 | 0x40 | In the beginning God created the heaven and the earth. |
| 0x3F3F40 | 2 | Start address of code |
| 0x3F3F42 | 2 | End address of code |
| 0x3F3F44 | 2 | Adler-32 Checksum |
| 0x3F3F46 | 2 | Do not check checksum and verse if this field is 0x12345678  Meant so that code may be burned by debugger and still run, with the CCS environment unaware of the statistics. |
| 0x3F3F80 | 2 | CAN address of device - Burner |
| 0x3F3F82 | 2 | CAN address of device – J1939 |
| 0x3F3F84 | 0x3c | Other management data TBD |
| 0x3F4000 | 2 | (Reserved for unused) Serial number of the device. |

**Checksums**

Checksums are according to the Adler32 algorithm (https://en.wikipedia.org/wiki/Adler-32). This algorithm is preferred over standard checksum as it accounts not only for the values in the code but also their order. The following is an excerpt from the above Wikipedia link:

**Calculation**

An Adler-32 checksum is obtained by calculating two [16-bit](https://en.wikipedia.org/wiki/16-bit) checksums *A* and *B* and concatenating their bits into a 32-bit integer. *A* is the sum of all [bytes](https://en.wikipedia.org/wiki/Byte) in the stream plus one, and *B* is the sum of the individual values of *A* from each step.

At the beginning of an Adler-32 run, *A* is initialized to 1, *B* to 0. The sums are done [modulo](https://en.wikipedia.org/wiki/Modular_arithmetic) 65521 (the largest [prime number](https://en.wikipedia.org/wiki/Prime_number) smaller than 216). The bytes are stored in one 32bit unsigned integer, *B* occupying the two most significant bytes.

The function may be expressed as

*A* = 1 + *D*1 + *D*2 + ... + *Dn* (mod 65521)

*B* = (1 + *D*1) + (1 + *D*1 + *D*2) + ... + (1 + *D*1 + *D*2 + ... + *Dn*) (mod 65521)

= *n*×*D*1 + (*n*−1)×*D*2 + (*n*−2)×*D*3 + ... + *Dn* + *n* (mod 65521)

*Adler-32*(*D*) = *B* × 65536 + *A*

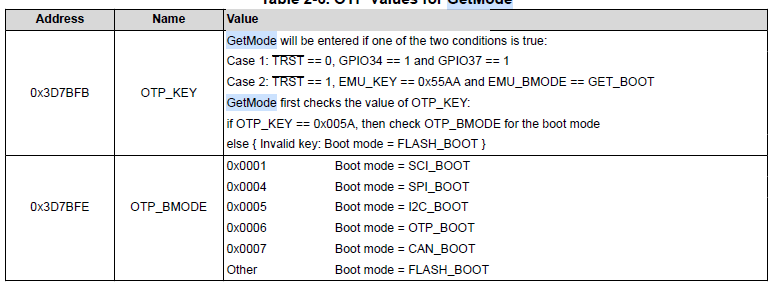
where *D* is the string of bytes for which the checksum is to be calculated, and *n* is the length of *D*.

# Appendix: Manual excerpts

## Boot behavior

Given that on reset wakeup the values are GPIO37 = 1 , GPIO34 = 1, the boot behaves by “GetMode()” – a function in the BootRom that scans OTP for further instructions.

The GetMode behavior is summarized below:



We assume that the OTP memory is pristine, so OTP\_KEY is not valid and the result shall be FLASH\_BOOT.

Excerpt from the Ref manual:

*Jump to flash is the default behavior of the Get Mode boot option. Jump to flash is also available as an*

*emulation boot option.*

*In this mode, the boot ROM software configures the device for C28x operation and branches directly to*

*location 0x3F 7FF6. This location is just before the 128-bit code security module (CSM) password locations.*

*You are required to have previously programmed a branch instruction at location 0x3F 7FF6 that will redirect*

*code execution to either a custom boot-loader or the application code.*

# TBD management

What is the default slave ID?

What is the information delivered as ECU name?