

Zubax Babel Datasheet

Zubax Robotics

Akadeemia rd. 21/1, Tallinn 12618, Estonia

info@zubax.com

Q&A: forum.zubax.com

Revision 2019.02.20

Overview

Zubax Babel is an advanced USB-CAN and UART-CAN adapter that can be used as a standalone device or as an embeddable module for OEM¹.

Babel uses the quasi-standard SLCAN (aka LAWICEL) protocol (with Zubax extensions) for transferring CAN data over USB and UART. There is a wide selection of software products that can communicate with SLCAN-compatible adapters.

Features

- Very low latency – the cumulative latency between the host system and the CAN bus is under 1 millisecond.²
- Very high throughput – the device handles more than 5000 frames per second in either direction continuously.³
- Large RX buffer (255 CAN frames plus 2KB of serial buffers) allows the device to handle short-term traffic bursts regardless of the throughput of the host-side interface.
- Proper prioritization of the outgoing CAN frames. The adapter properly schedules the outgoing frames, avoiding the inner priority inversion problem in the TX queue.
- Embedded 120 Ω CAN bus termination resistor that can be turned on and off by a command.
- Embedded CAN bus power supply output that can be turned on and off by a command.
- Hardware RX/TX timestamping.
- CAN bus supply voltage monitoring.
- SMD soldering pads for OEM applications.

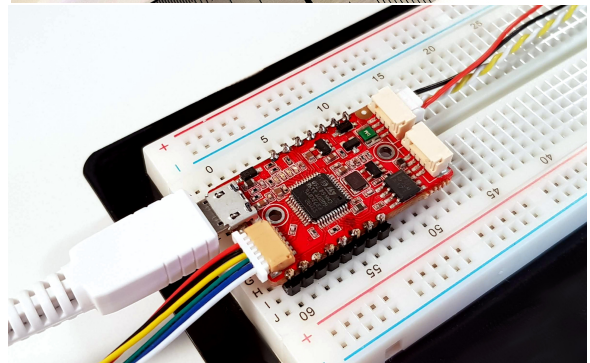
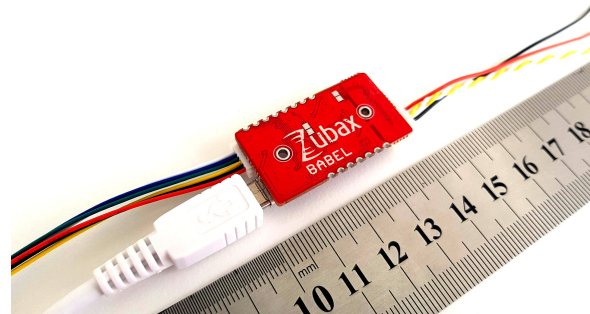
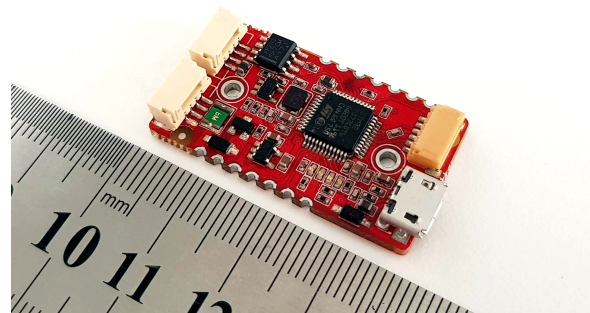
¹Original equipment manufacturer.

²Tested via USB on Linux 3.13 using a low-latency SLCAN driver from the PyUAVCAN library.

³Tested via USB on Linux 3.13. When using the UART interface, the throughput is limited by the UART baud rate setting.

Applications

- General-purpose USB-CAN or UART-CAN adapter.
- Diagnostic, monitoring, and development tool for **UAVCAN** networks. We recommend the UAVCAN GUI Tool for use with UAVCAN applications.
- Generic CAN/UAVCAN development board.
- Programmable CAN unit in OEM applications.



List of figures

1.1 Plastic enclosure.	1
2.1 Power supply architecture.	4
2.2 CAN bus interconnection diagram.	5
2.3 Physical dimensions of Babel.	9
2.4 Babel pinout.	10
6.1 Using the UAVCAN GUI Tool for configuration parameter management.	25
7.1 Bootloader state machine.	28
A.1 An overview of the SLCAN API implemented in various SLCAN-compatible adapters.	33

List of tables

2.1 Absolute maximum ratings.	3
2.2 Environmental conditions	3
2.3 Power supply summary	4
2.4 Power supply characteristics	4
2.5 UAVCAN Micro (JST GH) standard connector pinout	6
2.6 CAN bus interface characteristics	6
2.7 Dronecode Debug Mini standard connector pinout	7
2.8 Dronecode debug port characteristics	8
2.9 SMD signal pad characteristics	8
2.10 Physical characteristics	9
3.1 LED indicators	12
3.2 Status LED behavior	12
4.1 CAN controller configuration commands	14
4.2 CAN bit rate setting interpretation	14
4.3 CAN frame transmission commands	15
4.4 CAN frame transmission responses	15
4.5 Miscellaneous commands	16
4.6 UART baud rate setting interpretation	16
4.7 Bits of the F bit mask	17
4.8 SLCAN notifications	17
4.9 SLCAN notification example	18
4.10 Zubax ID fields	21
6.1 Configuration parameter index	26
7.1 Bootloader communication interfaces.	27
7.2 Bootloader states	28
7.3 Bootloader state indicated via the CAN traffic LED indicator.	28
7.4 Bootloader error codes	29
7.5 Zubax ID fields	31

1 Overview

Zubax Babel is an advanced USB-CAN and UART-CAN adapter that can be used as a standalone device or as an embeddable module for OEM.

1.1 Accessories

Babel can be used with the following accessories:

- Plastic enclosure described in the section 1.1.1.
- UAVCAN cabling and related items.
- Standard USB cables.
- Cables compatible with the Dronecode Autopilot Connector Standard.

Please contact your supplier for the ordering information.

1.1.1 Enclosure

Babel can be enclosed in a plastic enclosure pictured on the figure 1.1. The enclosure provides a mechanical protection that makes the device more suitable for use as a general-purpose desktop CAN adapter tool.



Figure 1.1: Plastic enclosure.

Please contact your supplier for the ordering information; alternatively, visit the website at https://github.com/Zubax/zubax_babel to download the 3D-printable enclosure models suitable for in-house manufacturing.

1.2 Quality assurance

Every manufactured Babel undergoes an automated testing procedure that validates that the device is functioning as designed. The test log for every manufactured device is available on the web at https://device.zubax.com/device_info. This feature can be used to facilitate the traceability of purchased devices and provide additional safety assurances.

Every manufactured device has a strong digital signature stored in its non-volatile memory which

proves the origins of the product and eliminates the risk of sourcing unlicensed or counterfeit hardware. This signature is referred to as Certificate of Authenticity (CoA). Please refer to the [Zubax Knowledge Base](#) to learn more about the certificate of authenticity and how it can be used to trace the origins of your hardware.

2 Characteristics

2.1 Absolute maximum ratings

Stresses that exceed the limits specified in this section may cause permanent damage to the device. Proper operation of the device within the limits specified in this section is not implied.

Table 2.1: Absolute maximum ratings

Symbol	Parameter	Min	Max	Unit
V_{supply}	Supply voltage	-0.3	6	V
T_{oper}	Operating temperature	-40	85	°C
	UART RX input voltage	-0.3	6	V
	CAN H/L input voltage	-4	16	V

2.2 Environmental conditions

Table 2.2: Environmental conditions

Symbol	Parameter	Min	Max	Unit
T_{oper}	Operating temperature	-40	85	°C
T_{stor}	Storage temperature	-40	85	°C
ϕ_{oper}	Operating humidity ^a	0	100	%RH

^a Condensation not permitted.

2.3 Power supply

Babel requires a +5 V DC power supply input which can be delivered via any of the available interface connectors.

The device has a reverse current protection on the USB power input which prevents back-powering the USB host when it is turned off.

The CAN power output has a solid state switch that can be enabled and disabled by setting the configuration parameter `can.power_on` (page 26). The device can always be powered from the CAN bus regardless of the state of the power switch.

An additional +3.3 V DC output on the SMD pads can be used to power any external circuitry when the device is used as a development board or in OEM applications. The pinout specification for the SMD pads is provided in the section 2.4.4.

The topology of the power supply circuits is documented on the figure 2.1. Power input and output capabilities per interface are summarized in the table 2.3.

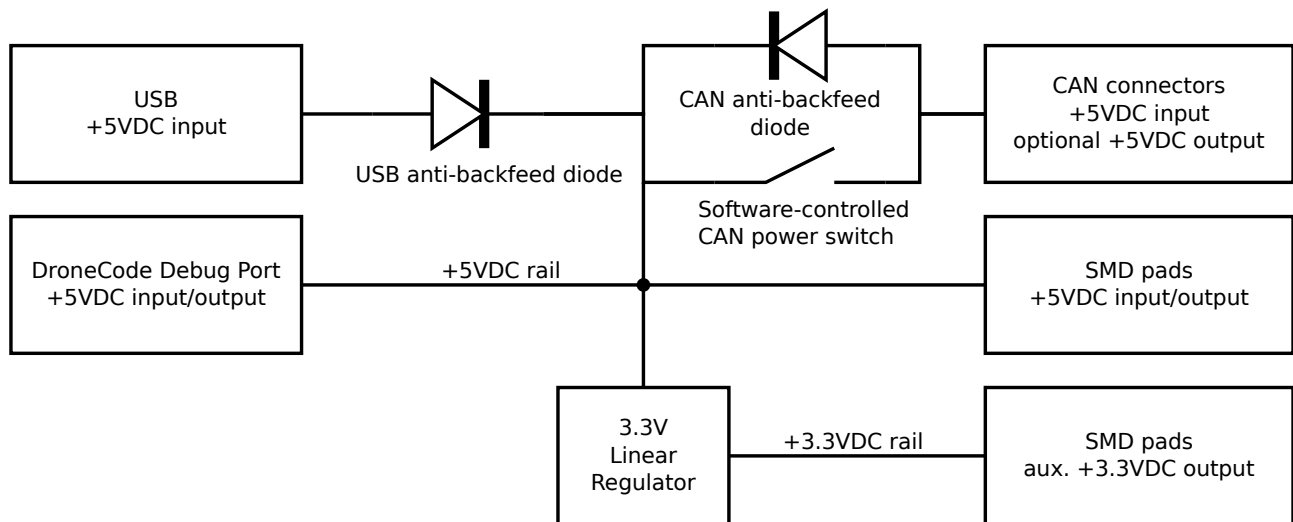


Figure 2.1: Power supply architecture.

Table 2.3: Power supply summary

Interface	Direction	Section
CAN	Input, optional output	2.4.1
USB	Input	2.4.2
Dronecode debug port	Input, output	2.4.3
SMD pads	Input, output	2.4.4

Table 2.4: Power supply characteristics

Symbol	Parameter	Min	Typ	Max	Unit
V_{supply}	Supply voltage ^a	4.0	5.0	5.5	V
I_{supply}	Self power consumption ^b	30	50	80	mA
I_{total}	Total power consumption ^c			500	mA
	3.3 V rail output voltage	3.2	3.3	3.4	V
	3.3 V rail external load			100	mA

^a Any power input.

^b SMD pads floating, Dronecode port disconnected.

^c Maximum power that the device can consume, including all dependent power consumers, such as the CAN bus power line or external loads connected via the SMD pads.

2.4 Communication interfaces

Babel implements the following set of communication interfaces:

- CAN 2.0A/B (ISO 11898-2).
- USB port (CDC ACM, virtual serial port).
- Dronecode debug port.
- SMD pads for OEM applications.

2.4.1 CAN bus

Babel implements the ISO 11898-2 CAN 2.0 A/B bus physical layer standard, also known as high-speed CAN. The CAN bus interface is equipped with two standard UAVCAN Micro connectors

(JST GH)⁴ electrically parallel to each other, which facilitates easy integration of the device into the end application without the need to use T-connectors.

The default bit rate of the CAN bus can be set via the configuration parameter `can.bitrate` (page 26), in bits per second.

Babel always attempts to configure the CAN timings so as to achieve settings as close as possible to the following:

- Sampling point location: 87.5%.
- Time quanta per bit: 8–10 for bit rate ≥ 1 Mbps, 16–17 otherwise.
- Resynchronization jump width: 1 time quantum.

The CAN interface recovers from the bus-off state automatically once the controller has observed 128 occurrences of 11 consecutive recessive bits on the bus, as defined by the CAN specification.

The device has an embedded $120\ \Omega$ CAN termination resistor that can be enabled and disabled by setting the configuration parameter `can.terminator_on` (page 26). Changes to this configuration parameter take effect immediately.

The power switch, when turned on, delivers a +5 V DC supply to the CAN bus, which can be used to power other CAN bus nodes from Babel (section 2.3). The power switch is controlled by the configuration parameter `can.power_on` (page 26). Changes to this configuration parameter take effect immediately.

The physical locations of the CAN connectors are documented in the section 2.6.

2.4.1.1 Device interconnection

The figure 2.2 shows a typical CAN bus topology. Observe that if Babel is a last device on the bus, a separate termination resistor would not be required, since Babel has an embedded termination resistor that can be enabled by setting the configuration parameter `can.terminator_on` (page 26).

The CAN bus interface of Babel is not redundant. If redundancy is desired, multiple Babels should be used in parallel, one per bus.

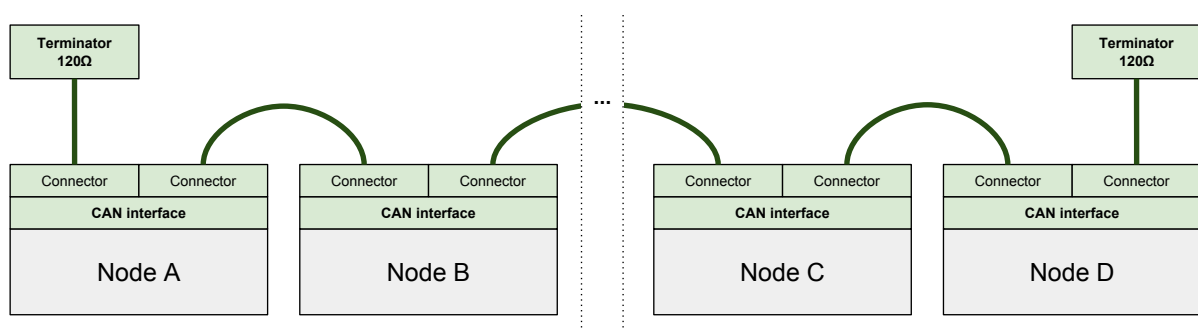


Figure 2.2: CAN bus interconnection diagram.

⁴<https://kb.zubax.com/x/EoAh>

2.4.1.2 Characteristics

Table 2.5: UAVCAN Micro (JST GH) standard connector pinout

Pin no.	Type	Function
1	Power	+5 V power supply output and/or pass-through ^a
2	Input/output	CAN High
3	Input/output	CAN Low
4	Ground	Power & signal ground

^a Power output can be enabled/disabled by setting the configuration parameter `can.power_on` (page 26).

Table 2.6: CAN bus interface characteristics

Symbol	Parameter	Min	Typ	Max	Unit
	Bit rate	10		1000	Kbps
	Positive-going input threshold voltage		750	900	mV
	Negative-going input threshold voltage	500	600		mV
	Differential output voltage, dominant	1.5	2.0	3.0	V
	Differential output voltage, recessive	-120	0	12	mV
	Inter-connector current pass-through ^a	-1		1	A
	Connector resistance during the device's lifetime		30	50	mΩ
	CAN bus power output voltage ^b	$V_{\text{supply}} - 0.4$	$V_{\text{supply}} - 0.3$	V_{supply}	V
	CAN bus power output current	-1		0.3	mA
	Resistance of the embedded CAN terminator	113	120	127	Ω

^a The limit is imposed by the PCB.

^b CAN power supply voltage is a function of the input supply voltage and the voltage drop in the CAN power switch circuit. The latter is dependent on the CAN power supply current.

2.4.2 USB

The device implements a full-speed USB 2.0 port with the standard CDC ACM interface (also known as “virtual serial port”). The device features driverless compatibility with all major operating systems (Windows, GNU/Linux, Mac OS).⁵

The physical connector type is USB micro B (which is one of the most common device-side USB connector types).

2.4.2.1 Identification

Babel will report the following properties to the USB host:

- Vendor ID – 0x1D50
- Product ID – 0x60C7
- Vendor string – Zubax Robotics
- Device description string – Zubax Babel
- Device ID – the 128-bit globally unique device ID (section 2.5) as a hexadecimal string

⁵Get more knowledge and helpful tips at <https://kb.zubax.com>.

2.4.3 Dronecode debug port

The device features a Dronecode debug port interface available via the standard Dronecode Debug Mini connector (DCD-Mini)⁶. This port can be conveniently used with the **Zubax Dronecode Probe**, or any other UART-capable hardware with a compatible connector.

The physical location of the connector is documented in the section 2.6.

The Dronecode debug port provides access to the UART and JTAG/SWD interfaces; the latter is mostly useful for OEM applications and for using Babel as a development board.

2.4.3.1 UART interface

The baud rate can be changed by setting the configuration parameter `uart.baudrate` (page 26). New baud rate settings take effect immediately.

The following parameters of the UART interface are fixed and cannot be changed by the user:

- Word size – 8 bit
- Parity control – none
- Stop bits – 1 bit

2.4.3.2 SWD interface

The SWD interface is a standard debug interface implemented in many ARM cores. Please refer to the specialized literature for additional information.

Information about the use of Babel in OEM applications is provided in the section 5.

The recommended SWD adapter for use with Babel is the **Zubax Dronecode Probe**.

2.4.3.3 Characteristics

Table 2.7: Dronecode Debug Mini standard connector pinout

Pin no.	Type	Name	Comment
1	Power	TPWR	+5 V power supply input/output
2	Output	UART_TX	
3	Input	UART_RX	Pulled up with a resistor
4	Input/Output	SWDIO	For OEM & development use
5	Input	SWDCLK	For OEM & development use
6	Ground	GND	Power & signal ground

⁶<https://wiki.dronecode.org/workgroup/connectors/start>

Table 2.8: Dronecode debug port characteristics

Symbol	Parameter	Min	Typ	Max	Unit
	Supported UART baud rates	2400	115200	3 000 000	baud/s
	Low-level input voltage	-0.3	0	1.6	V
	High-level input voltage	2.1	3.3	5.5	V
	Low-level output voltage	0	0	0.5	V
	High-level output voltage	2.8	3.3	3.4	V
	Source/sink current via data pins			10	mA
	UART RX pull up resistance	30	40	50	k Ω
	Connector resistance during device lifetime		20	40	m Ω

2.4.4 SMD pads

Babel exposes a set of SMD pads which facilitate the following applications of the device:

- Programmable CAN module for OEM applications. Babel can be directly soldered into a larger PCB using the exposed SMD pads.
- CAN development board. Standard 2.54 mm connectors can be soldered to the SMD pads in order to make the device compatible with standard prototyping breadboards.
- Generic USB/UART GPIO controller (section 4.4.1.6).

Information about the use of Babel in OEM applications is provided in the section 5. The pinout specification for the SMD pads is provided on the figure 2.4.

Table 2.9: SMD signal pad characteristics

Symbol	Parameter	Min	Typ	Max	Unit
	Low-level input voltage	-0.3	0	1.6	V
	High-level input voltage	2.1	3.3	5.5	V
	Low-level output voltage	0	0	0.5	V
	High-level output voltage	2.8	3.3	3.4	V
	Pull-down or pull-up resistance	25	40	55	k Ω
	Source/sink current (magnitude)			10	mA

2.5 Product identification

This section documents the device properties that are reported in response to identification requests, such the CLI command `zubax_id` (section 4.4.1.2).

The product ID string is reported as “com.zubax.babel”. The prefix “com.zubax.” is shared by many of the products designed by Zubax Robotics.

Every manufactured device has a globally unique 128-bit ID (UID) that cannot be changed.

Every manufactured device is equipped with a certificate of authenticity, which is a function of, among other things, the UID and the product ID of the device. Please refer to the web resources provided by Zubax Robotics to learn more about the certificate of authenticity and how it can be used to verify the authenticity of products.

2.6 Physical characteristics and pinout

The figure 2.3 documents the basic mechanical characteristics of Zubax Babel, such as the placement of connectors and mounting holes. The pinout of the SMD pads and other connectors is shown on the figure 2.4.

Table 2.10: Physical characteristics

Symbol	Parameter	Typ	Unit
m	Mass	4	g

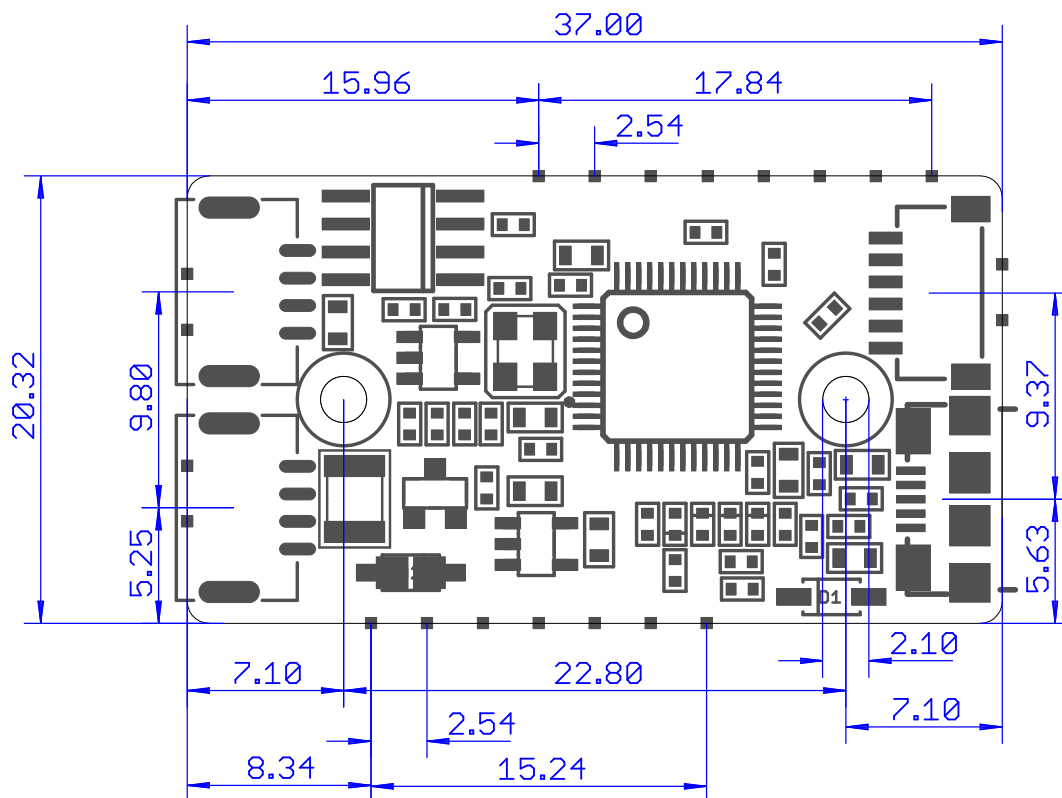


Figure 2.3: Physical dimensions of Babel.

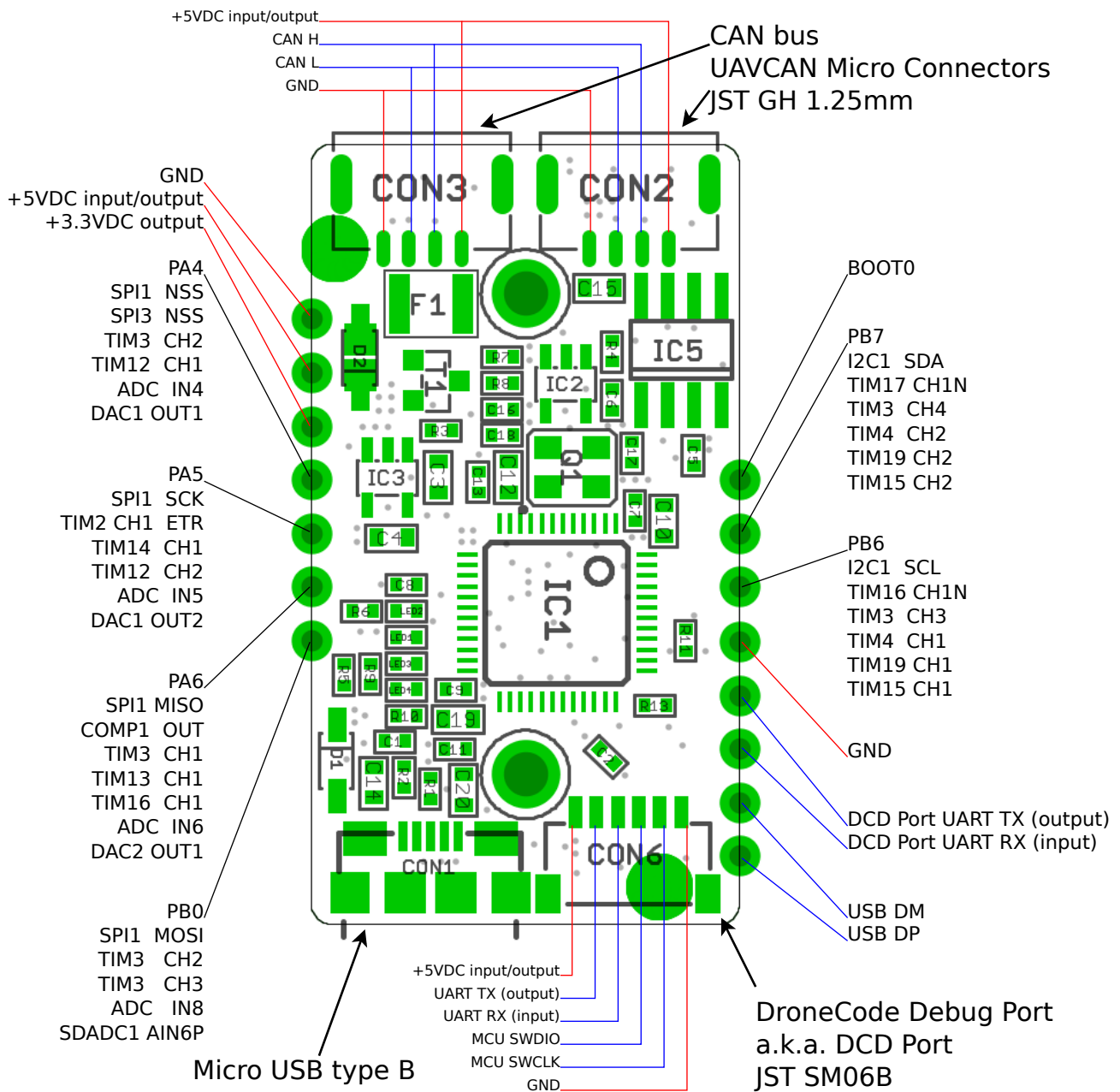


Figure 2.4: Babel pinout.

3 Operating principles

3.1 Overview

Babel keeps its CAN controller disabled by default in order to ensure that no unnecessary interference with the CAN bus is introduced. The CAN controller is enabled only if the host has requested Babel to open the data tunnel. This is reviewed in more detail in the section [4](#).

When opening the data tunnel, it is possible to explicitly specify the desired bit rate of the CAN bus. If the CAN bit rate was not specified explicitly, Babel will use the value stored in the parameter `can.bitrate` (page [26](#)). If the bit rate was specified explicitly, the parameter value will be automatically updated.

When the tunnel is opened, all of the buffers become flushed (i.e. cleared) and the interface statistic counters get reset. An attempt to open the tunnel when it is already opened is interpreted as if there was an implicit close command before the tunnel is re-opened again.

Babel contains two RX buffers connected in series: a hardware buffer that can contain up to 3 CAN frames, and a software buffer that can contain up to 255 CAN frames. The specifics of RX buffering are explained in the subsequent chapters of this document.

The capacity of the software TX buffer is 100 frames. The TX logic is not equipped with hardware buffers.

Babel runs a single instance of an SLCAN protocol handler that can be attached either to the USB CDC ACM interface (virtual serial port) or to the physical UART interface. By default, the UART interface is used. If Babel detects that the USB interface is connected to a USB host, it disconnects the SLCAN protocol handler from the UART port and attaches it to the USB CDC ACM port. The reverse happens when the USB port becomes disconnected. It is therefore impossible to use both USB and UART interfaces at the same time concurrently.

3.2 Start up and initialization

Immediately after powering on, the device starts the embedded bootloader (described in detail in the section [7](#)). The bootloader awaits for external commands for a few seconds. If no commands requesting it to download a new firmware image or to wait longer were received, and if a valid application (i.e. firmware) was found in the ROM, the bootloader starts the application. If no valid application is found in the ROM, the bootloader will wait for commands forever.





3.3 GPIO pins

The device allows the user to control some of its SMD-exposed GPIO pins via CLI (section [4.4.1.6](#)). The specified GPIO configuration survives until the device is restarted. Upon restart, the default configuration is applied, which is identical for all GPIO pins: discrete input with pull-down.

3.4 LED indicators






Babel is equipped with four separate LED indicators that reflect the current state of the device. Their functions are summarized in the table [3.1](#).

Table 3.1: LED indicators

Color	Name	Behavior
 Red	CAN power LED	Glowes when the CAN bus power output is enabled.
 Orange	CAN terminator LED	Glowes when the CAN bus termination resistor is enabled.
 Blue	Status LED	See table 3.2.
 Green	CAN traffic LED	Blinks once if at least one CAN frame was successfully transmitted or successfully received in the last 25 milliseconds. Glowes steadily when the intensity of CAN traffic is higher than 40 frames per second. The function of this LED is different while the bootloader is running; see the section 7.

The behavior of the status LED is more complex than that of other indicators. It is specified in the table 3.2. Observe that there is one special mode that is used while the embedded bootloader is running. The bootloader is documented separately in the section 7.

Table 3.2: Status LED behavior

Status	LED pattern (step 50 ms)	LED behavior
Bootloader is running		Glowing continuously. In this mode, the CAN traffic LED behaves as described in the section 7.
CAN channel closed		Turned OFF
CAN channel open, normal operation		Blinking 1 Hz
CAN channel open, error passive		Blinking 4 Hz
CAN channel open, bus off		Blinking 10 Hz

4 SLCAN protocol

4.1 Introduction

The SLCAN protocol (also known as LAWICEL protocol) is a quasi-standard protocol designed for tunneling of CAN data (such as CAN frames, commands, and adapter status information) via serial links (such as UART or USB virtual serial ports).

Babel implements all of the mandatory SLCAN commands, so that its compatibility with third party software products that rely on SLCAN is ensured. A brief recap of the standard SLCAN commands implemented in third-party products can be found in the appendix [A](#).

SLCAN is an ASCII text-based protocol, where the data is exchanged in blocks. A block may contain only printable ASCII characters.

The end of a block is marked either with the ASCII carriage return character (`\r`, code 13), which is interpreted as a positive acknowledgement (ACK); or the ASCII bell character (`\a`, code 7), which is interpreted as a negative acknowledgement (NACK).

A block begins with a well-defined character which indicates the kind of information the block is carrying; in this description we're going to refer to this character as *Block ID*.

The protocol defines three types of data blocks:

Command Blocks of this type are sent from the host to the adapter.

Response Blocks of this type are sent from the adapter to the host as a reaction to a command. All commands provide exactly one response, unless stated otherwise.

Notification Blocks of this type are generated by the adapter asynchronously.

All commands and notifications are always terminated with the ACK character (`\r`). A response will be terminated with ACK if the corresponding command has been executed successfully, and with NACK if the command has failed or if the command could not be understood by the adapter.

Babel also implements an extension on top of the SLCAN protocol that allows the host to execute arbitrary CLI commands over the same serial link that is used by SLCAN. This feature is documented in the section [4.4](#).

Note that some commands alter the configuration parameters of the adapter. All parameters are automatically stored in a non-volatile memory, and their values are restored automatically whenever the adapter is turned on. The section [6](#) provides an in-depth description of the configuration parameters and their non-volatile storage.

The SLCAN interface can be exposed either via USB or via UART, but not both at the same time. Babel connects the SLCAN interface to the USB virtual serial port as long as it is connected to a USB host. When the USB interface is disconnected, the SLCAN interface is available via UART.

A high quality host-side implementation of the SLCAN protocol in Python can be found in the [PyUAVCAN library](#) (MIT software license).

4.2 SLCAN commands

4.2.1 CAN controller configuration commands

Table 4.1: CAN controller configuration commands

Block ID	Arguments	Purpose
S	Section 4.2.1.1	Set the CAN bit rate. The CAN bit rate will be stored in the configuration parameter <code>can.bitrate</code> (page 26).
O (capital o)	None	Open CAN channel in normal mode; re-open if already open. The bit rate value will be taken from <code>can.bitrate</code> (page 26).
L	None	Open CAN channel in silent mode (listen only); re-open if already open. The bit rate value will be taken from <code>can.bitrate</code> (page 26). Section 4.2.1.2.
l (lowercase L)	None	Open CAN channel in normal mode with loopback enabled; re-open if already open. The bit rate value will be taken from <code>can.bitrate</code> (page 26). Section 4.2.1.3.
C	None	Close CAN channel; do nothing if the channel is not open.
M	Any	This command is not applicable to Babel, it is implemented only for compatibility reasons. Arguments are not validated.
m	Any	See M.

All commands return ACK (\r) on success and NACK (\a) on failure. Commands that are implemented only for compatibility always report success.

4.2.1.1 CAN bit rate configuration

The command `s` accepts a non-negative decimal number which represents the desired CAN bit rate. If the channel is open, changes in the bit rate configuration will not take effect until it is re-opened again. The values are interpreted as specified in the table 4.2.

Table 4.2: CAN bit rate setting interpretation

Value	Interpretation, bit/s
0	10000
1	20000
2	50000
3	100000
4	125000
5	250000
6	500000
7	800000
8	1000000
Any other	The number is interpreted as-is, no additional conversion is performed.

4.2.1.2 Silent mode

When the channel is open in the silent mode, the adapter configures its CAN controller in silent mode as well. This mode ensures that the adapter will not interfere with the CAN bus, regardless of the physical state of the interface (e.g. the controller will not confirm received CAN frames; transmission of any CAN frames, including error frames, will never take place).

In silent mode, all outgoing CAN frames are ignored by the adapter. CAN frames received from the bus are processed as usual.

4.2.1.3 Loopback mode

When the channel is open with loopback enabled, all transmitted frames will be immediately sent back to the host after they were successfully delivered to the CAN bus.

If the SLCAN flags are enabled, loopback frames will be marked with an appropriate flag. More information on SLCAN flags is provided in the section 4.3.

4.2.2 CAN frame transmission commands

The following documentation uses the specified below notation to document the arguments of SLCAN commands:

i A hexadecimal digit that encodes the identifier of the current CAN frame.

d A hexadecimal digit that encodes the DLC (data length code) of the current CAN frame.

***** An arbitrary sequence of useful bytes encoded as a hexadecimal string. For example, the sequence 0110FF encodes the following sequence of three bytes: 1, 16, 255.

In SLCAN, hexadecimal digits can use both uppercase and lowercase Latin letters; therefore, the complete set of characters that may occur in a hexadecimal number is as follows: 0123456789abcdef ABCDEF.

Table 4.3: CAN frame transmission commands

Block ID	Argument format	Transmitted frame	Example of a complete SLCAN block
T	iiiiiiiid*	Data frame with 29-bit ID	T0123456780102030405060708\r
t	iiid*	Data frame with 11-bit ID	t7FF0\r
R	iiiiiiiid	RTR frame with 29-bit ID ^a	R1234f00d8\r
r	iiid	RTR frame with 11-bit ID ^a	r008\r

^a RTR frames don't carry payload data.

All of the above listed commands may generate the responses specified in the table 4.4.

Note that if a frame could not be scheduled for transmission due to the TX buffer being full, the adapter would still return success. Use the SLCAN flags together with the loopback mode in order to be able to detect when outgoing frames are dropped by the adapter.

Table 4.4: CAN frame transmission responses

Response	Meaning
Z\r	The frame has been processed successfully (for 29-bit ID).
z\r	The frame has been processed successfully (for 11-bit ID).
\a	The adapter could not transmit the frame. Some of the possible reasons include: a malformed SLCAN block, the CAN channel is not open.

4.2.3 Miscellaneous commands

Table 4.5: Miscellaneous commands

Block ID	Arguments	Purpose
U	Section 4.2.3.1	Set the UART baud rate. The setting will be stored in the configuration parameter <code>uart.baudrate</code> (page 26).
Z	0 or 1	Enable or disable the RX and loopback timestamping. The provided value will be stored in the configuration parameter <code>slcan.timestamping_on</code> (page 26).
F	None	Get and clear the status flags.
V	None	Get the hardware and software version numbers.
N	None	Get the 128-bit unique ID of the device as a hexadecimal string (section 2.5).

4.2.3.1 UART baud rate configuration

The command U accepts a non-negative decimal number which represents the desired UART baud rate.

The changes will take effect shortly after the command is executed (typically within 100 milliseconds), no reboot is necessary; therefore, the host should adjust the baud rate of the serial port immediately after this command is executed.

The argument is interpreted as specified in the table 4.6.

Table 4.6: UART baud rate setting interpretation

Value	Interpretation, baud/s
0	230400
1	115200
2	57600
3	38400
4	19200
5	9600
6	2400
Any other	The number is interpreted as-is, no additional conversion is performed.

At least the following baud rates are supported by the UART interface: 2400, 9600, 19200, 38400, 57600, 115200 (this is the default), 230400, 460800, 921600, 1 000 000, 3 000 000.

4.2.3.2 CAN frame timestamping

The command Z can be used to enable and disable the CAN frame timestamping feature.

The state of the timestamping feature is kept in the configuration parameter `slcan.timestamping_on` (page 26).

The changes will take effect shortly after the command is executed (typically within 100 milliseconds), no reboot is necessary.

4.2.3.3 Reading and clearing the status flags

The command F produces the following response:

| F??\r

Where ?? is a hexadecimal bit mask. The meaning of each bit in the bit mask is documented in the table 4.7. Bits that are not used should be ignored when processing the bit mask.

Table 4.7: Bits of the F bit mask

Bit	2 ^{Bit}	Name	Meaning
0	1		Not used.
1	2		Not used.
2	4		Not used.
3	8	RX overrun	The RX queues (software, hardware, or both) have overflowed at least once since the last invocation of the F command or since the channel was opened.
4	16		Not used.
5	32	Error passive	The CAN error counter values are above the error passive limit (refer to the CAN bus specification for details).
6	64		Not used.
7	128	Bus off	The CAN controller is in the bus off state (refer to the CAN bus specification for details).

4.2.3.4 Requesting the version information

The command V produces the following response:

| V????\r

Where the fields are one-digit hexadecimal numbers with the following meanings, in that order:

- Hardware version, major.
- Hardware version, minor.
- Software version, major.
- Software version, minor.

4.3 SLCAN notifications

Babel emits SLCAN notifications in the following cases:

- A CAN frame is received.
- If loopback is enabled: a CAN frame has been successfully delivered to the bus.

The format of notifications is the same as for the CAN transmission commands, as described in the table 4.8.

Table 4.8: SLCAN notifications

Block ID	Argument format	Purpose
T	iiiiiiiid*	Received or successfully transmitted a 29-bit data frame.
t	iiid*	Received or successfully transmitted an 11-bit data frame.
R	iiiiiiiid	Received or successfully transmitted a 29-bit RTR frame.
r	iiid	Received or successfully transmitted an 11-bit RTR frame.

Obviously, notifications for transmitted frames will be emitted only if the channel is open in the loopback mode.

Frame notifications may be extended with timestamps and/or with flags, depending on the values of the configuration parameters `slcan.timestamping_on` (page 26) and `slcan.flags_on` (page 26), respectively.

When timestamping is enabled, every emitted CAN frame notification will be appended with four more hexadecimal characters containing the time, in milliseconds, when the frame was received. The millisecond timestamp overflows every 60 000 milliseconds (once a minute), so the valid values lie in the range from 0 to 59 999, inclusive.

The frame timestamp can be converted into the target clock domain, e.g. the monotonic clock of the host system, using the Olson algorithm⁷ (the timestamp synchronization feature in PyUAVCAN is based on the Olson algorithm as well).

In the loopback mode with timestamping enabled loopback frame notifications will carry the timestamp of the moment when they were delivered to the bus. This is sometimes called “TX timestamping”.

When SLCAN flags are enabled, frame notifications will be amended as listed below. Note that flags are always added at the end of an SLCAN block.

- Loopback frames will be appended with the character L at the end of the block. This flag allows the host to distinguish frames received from the bus from transmitted frames that were looped back.
- Other flags may be added in future revisions of the protocol.

For example, an outgoing frame T12345678401234568 will be looped back as shown in the table 4.9, depending on the configuration.

Table 4.9: SLCAN notification example

Flags?	Timestamping?	Representation	Comment
No	No	T12345678401234568\r	No changes.
No	Yes	T123456784012345680BED\r	Transmission timestamp 3053 milliseconds.
Yes	No	T12345678401234568L\r	Added flag L.
Yes	Yes	T123456784012345680BEDL\r	Both of the above two, flag after the timestamp.

4.4 CLI extensions

Babel implements a proprietary extension of the SLCAN protocol that allows it to run a conventional CLI⁸ shell over the same serial port that is used for SLCAN communications, while maintaining full compatibility with SLCAN.

A CLI command is a sequence of printable ASCII characters terminated with the `\r\n` sequence (ASCII codes 13, 10). The first non-whitespace character of a CLI command sequence cannot be also a valid SLCAN block ID⁹. The last limitation is necessary to ensure that the adapter can unambiguously distinguish between a CLI command and an SLCAN block at the time of reception of the first character. The early detection of SLCAN blocks allows the protocol state machine to dispatch CAN frames with minimal latencies.

Every CLI command returns a response that begins with the exact copy of the received command terminated with `\r\n`, then followed by an arbitrary number of lines of text, each terminated with `\r\n`, and finalized with the ASCII END-OF-TEXT character (code 3) immediately followed by the final `\r\n`.

⁷ “A Passive Solution to the Sensor Synchronization Problem”, Edwin Olson, 2010.

⁸ Command line interface.

⁹ For example, “T12345678401234568” is not a valid CLI command, but “fooT12345678401234568” is acceptable.

For example, a command “`cfg list`” (the excessive spaces were added for the purposes of demonstration) may produce the following response (in this example, the non-printable characters are shown with escape sequences, e.g. `\r`):

```
cfg list \r\n
can.bitrate      = 1000000      [10000, 1000000] (1000000)\r\n
can.power_on     = 0           [0, 1] (1)\r\n
can.terminator_on = 0          [0, 1] (1)\r\n
slcan.timestamping_on = 1      [0, 1] (1)\r\n
slcan.flags_on   = 0           [0, 1] (0)\r\n
uart.baudrate    = 115200      [2400, 3000000] (115200)\r\n
\x03\r\n
```

Where `\x03` is the ASCII END-OF-TEXT character.

The following properties of the protocol allow both the adapter and the host distinguish SLCAN data from CLI data in real time:

- CLI commands and their responses are terminated with `\r\n` rather than plain `\r`.
- A valid SLCAN command cannot be also a valid CLI command.

A compliant SLCAN driver that is capable of dealing with CLI extensions with virtually zero performance penalty can be found in the PyUAVCAN library.

CLI commands can be executed manually by connecting to the SLCAN port using a terminal emulator program. In this case it is recommended to enable local echo in the terminal emulator program, because the SLCAN protocol does not provide remote echo.

4.4.1 CLI commands

This section documents the implemented CLI commands and provides usage instructions for them.

4.4.1.1 *cfg*

This command provides access to the configuration parameter storage. The configuration parameters and their non-volatile storage are described in detail in the chapter 6.

Syntax:

- `cfg list` — list all configuration parameters, their current values, acceptable value intervals, and default values.
- `cfg save` — save the current configuration into the non-volatile storage. This command is redundant and normally need not be used, because firmware revisions starting from v1.1 commit all configuration into the non-volatile storage automatically upon modification.
- `cfg erase` — erase the current configuration and reset the non-volatile memory to the factory defaults.
- `cfg set <name> <value>` — assign the configuration parameter named `<name>` the value `<value>`. For example: `cfg set foo 42`. The non-volatile storage will be updated automatically.

The syntax `cfg list` prints one parameter per line, where the line is formatted as follows:

```
name = value [min, max] (default)
```

The number of whitespaces between tokens may vary.

Floating point parameters are always reported with the point symbol (`.`), which allows one to distinguish them from integer and boolean typed parameters.

Boolean parameters are reported as integers, where 1 represents the logical true and 0 represents the logical false. They can be distinguished from integer parameters by their minimum and maximum values being 0 and 1, respectively.

Whenever a parameter is assigned a new value, the device verifies if the new value is within the acceptable limits. If it is, the new value is assigned. Otherwise, the old value is retained. Afterwards the device prints out the resulting value of the parameter in the following format:

```
name = value
```

The number of whitespaces between tokens may vary.

The response can be used to check whether the new value was accepted by the device or not.

4.4.1.2 *zubax_id*

This command is available in all Zubax products that implement a command-line interface. It reports the complete information identifying this particular product: type, version information, make and model. The information is printed in a machine-readable yet human-friendly YAML¹⁰ format.

An example output is shown in the following listing. The meaning of each well-defined field is explained in the table 4.10. Note that the ordering of the fields is not guaranteed to be constant; furthermore, additional fields may be added in future firmware revisions.

```
product_id   : 'com.zubax.babel'
product_name : 'Zubax Babel'
sw_version   : '1.0'
sw_vcs_commit: 48923790
sw_build_date: Jun 20 2016
hw_version   : '1.0'
hw_unique_id : '0AAyAA1XMkEzNjkgAAAAAA=='
hw_info_url  : http://device.zubax.com/device_info?uid=380032000D5732413336392000000000
hw_signature : 'SknsmA7XugU9pF/+NN0oU26Gdq9Vvh0301Cw2qim17RXsU8y0ISoK0dMIh4QIHtXr36sMfx
H397RlSNc0TtWDPOyA713z0k+v+ZY5PGXkRFiUfspnU/EJ8+r0ur12dYp7NApx4l0kl0gNgHrGCA6lPxA8UqoW
9jdqaASuqPFZKg='
```

¹⁰<https://en.wikipedia.org/wiki/YAML>

Table 4.10: Zubax ID fields

Field name	Meaning
product_id	Product type identifier string. The same string is reported via UAVCAN as the node name string.
product_name	Human-readable product name.
sw_version	Firmware version number in the form “major.minor”.
sw_vcs_commit	Firmware version short commit identifier (e.g. short git commit hash). The abbreviation VCS stands for “version control system”. This number allows to pinpoint the exact revision of the firmware that is currently running.
sw_build_date	Firmware build date in the form “mmm dd yyyy”.
hw_version	Hardware version number in the form “major.minor”.
hw_unique_id	The 128-bit unique ID of this specific hardware instance (section 2.5). The UID is represented either as a hexadecimal string, or as a Base64 encoded string.
hw_signature	The certificate of authenticity (CoA) of this specific hardware instance encoded in a Base64 string. If this data is missing, please inform Zubax Robotics as soon as possible.
hw_info_url	Link to the web page that contains the test report, origin information, traceability data, and other important information about this specific hardware instance provided by the vendor. If this data is missing, or if the linked web page is unreachable, please inform Zubax Robotics as soon as possible.

4.4.1.3 stat

Returns a YAML dictionary containing the immediate state information of the adapter.

The statistic counters are reset every time the channel is opened. Note that the statistics are kept intact after the channel is closed.

At least the following fields are reported in the output:

open Whether the CAN channel is open, either true or false.

state One of the following: `error_active`, `error_passive`, `bus_off`. Please refer to the CAN protocol specification for the description of each state.

receive_error_counter CAN receive error counter. Refer to the CAN protocol specification for more information.

transmit_error_counter CAN transmit error counter. Refer to the CAN protocol specification for more information.

errors The number of CAN protocol errors reported by the CAN controller hardware.

bus_off_events The number of registered bus off events.

sw_rx_queue_overruns The number of overruns of the software RX queue. The software RX queue is positioned after the hardware RX queue.

hw_rx_queue_overruns The number of overruns of the hardware RX queue. The hardware RX queue is positioned before the software RX queue.

frames_tx The number of CAN frames successfully delivered to the bus. Frames that were dropped due to overruns of the TX queue do not affect this statistic.

frames_rx The number of CAN frames received from the bus. This statistic also includes frames that were lost due to overruns of the software RX queue. Frames that were lost due to overruns of the hardware RX queue do not affect this statistic.

tx_queue_capacity The capacity of the TX queue.

tx_queue_peak_usage The peak usage of the TX queue.

rx_queue_capacity The capacity of the RX queue.

rx_queue_peak_usage The peak usage of the RX queue.

tx_mailbox_peak_usage The maximum number of TX mailboxes that were used simultaneously by the driver. The adapter uses an advanced algorithm of TX mailbox assignment in order to eliminate the inner priority inversion problem.

bus_voltage The voltage of the CAN bus power supply line, in volts.

An example output of this command is shown below:

```
open           : true
state          : error_active
receive_error_counter : 0
transmit_error_counter: 0
errors         : 0
bus_off_events : 0
sw_rx_queue_overruns : 0
hw_rx_queue_overruns : 0
frames_tx      : 0
frames_rx      : 0
tx_queue_capacity : 100
tx_queue_peak_usage : 0
rx_queue_capacity : 255
rx_queue_peak_usage : 0
tx_mailbox_peak_usage : 0
bus_voltage    : 4.8
```

4.4.1.4 *bootloader*

This command reboots the device into the bootloader mode, described in the section 7. It can be used to initiate a firmware update procedure over USB or UART.

When the bootloader is started using this command, it will not boot the application automatically upon expiration of the boot timeout.

4.4.1.5 *reboot*

Unconditionally reboots the device.

4.4.1.6 *gpio*

Controls the GPIO pins exposed on the SMD pads. This command is available since firmware v1.2.

Syntax:

- `gpio <pin>` — read the current status of the pin without changing its mode.
- `gpio <pin> <mode>` — apply the specified mode and then read the status of the pin.

Where `pin` can be one of the following (see the pinout diagram 2.4) (case sensitive):

- pa4
- pa5
- pa6
- pb0
- pb6
- pb7

The `mode` argument, if specified, can be one of the following (case sensitive):

- `oh` — discrete output at high level.
- `ol` — discrete output at low level.
- `ih` — discrete input with pull-up.
- `il` — discrete input with pull-down.

Upon successful execution, the command returns either `h` or `l` (lowercase Latin L), depending on the current logical level of the selected pin.

The specified GPIO configuration survives until the device is restarted. After restart, the default GPIO configuration will be applied, as specified in the section [3.3](#).

5 OEM applications

Besides its primary role of a USB-CAN or UART-CAN adapter tool, Babel can be effectively used as an integral part of a larger system or a device: either as a CAN adapter or, if loaded with a custom user-developed firmware, in a completely different role. These use cases are referred to as OEM applications.

Additionally, Babel can be used as a prototyping platform for quick development of CAN-dependent (and especially UAVCAN-centric) applications.

More information about development of custom applications for Babel is available in the Zubax Knowledge Base¹¹.

5.1 SMD pads

It is expected that OEM applications may require Babel to be installed on a custom PCB. For this reason, Babel is given SMD pads spaced at the standard 2.54 mm pitch. Standard 2.54 mm pin connectors can be soldered to the SMD pads in order to enable easy integration with standard breadboards.

The signals that are routed to the SMD pads and other connectors are shown on the figure 2.4.

5.2 Custom applications

Please refer to Zubax Knowledge Base for detailed information about development of custom applications.

Zubax Babel is based on the microcontroller STM32F373CBT6; its main characteristics are outlined below.

- Core: ARM Cortex-M4F (with a hardware floating point unit) clocked at 72 MHz.
- Flash memory capacity: 128 KB.
 - First 32 KB are occupied by the bootloader and some auxiliary persistent data (which should never be erased by the user).
 - The following 94 KB are available for the user's application.
 - Last 2 KB are reserved for non-volatile configuration storage.
- RAM capacity: 24 KB.
 - First 256 bytes are reserved for the bootloader.
 - Remaining 24320 bytes are available for the user's application.

Custom applications can be loaded either directly via the SWD interface¹², in which case great care should be taken to not accidentally erase the bootloader; or via the bootloader itself. More information about the bootloader can be gathered in the section 7.

¹¹<https://kb.zubax.com>

¹²We recommend Zubax Dronecode Probe for this task.

6 Configuration parameters

Configuration parameters are stored in a non-volatile memory on the adapter. They can be modified by using the CLI and also by using dedicated SLCAN commands, as described in the section 4.

UAVCAN GUI Tool¹³ provides a convenient way of managing the Babel's configuration parameters.

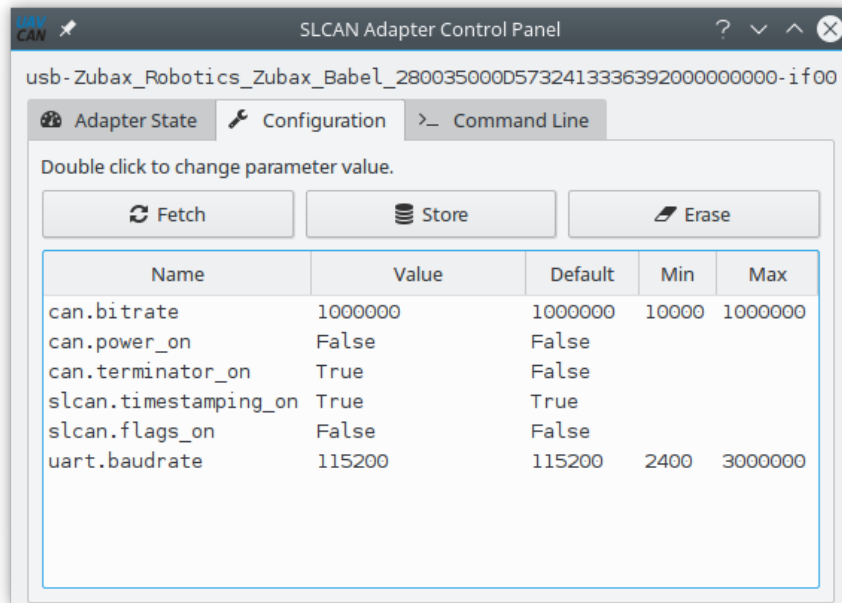


Figure 6.1: Using the UAVCAN GUI Tool for configuration parameter management.

6.1 Non-volatile configuration storage

All configuration parameters are stored in a non-volatile memory that retains its contents across power cycles.

Modification of any single parameter will trigger the device to commit all of them into the non-volatile memory.¹⁴

If the device is turned off while the configuration storage is being updated, the stored configuration data may get damaged.

The stored configuration is read from the non-volatile memory once upon boot-up. If the device detects that the stored configuration data has been damaged, it will automatically revert to the factory default configuration. Babel can always reliably detect damage of the stored configuration data, so it is guaranteed that an invalid configuration can never be loaded.

6.2 Firmware update considerations

The configuration parameter sets of different firmware revisions may be incompatible with each other. For instance, some configuration parameters may be added, removed, or their value intervals may be changed.

¹³<http://uavcan.org>

¹⁴The auto-save feature is available since firmware v1.1.

Babel always checks whether the stored configuration data is compatible with the current firmware revision. If it is detected that the stored configuration cannot be applied to the current version of the firmware, the device will automatically revert to the factory default configuration.

Keep these considerations in mind when updating the firmware.

6.3 Configuration parameter index

The minimum, maximum, and default values provided in the table are shown for exemplary purposes only, and they are *not expected to be valid* for all firmware revisions that this document applies to. Intervals and default values may change in newer revisions of the firmware or the hardware.

Table 6.1: Configuration parameter index

Name	SLCAN alias	Takes effect at	Pages	Min	Max	Def.	Description
can.bitrate	S	channel open	5, 11, 13	10 000	1 000 000	1 000 000	CAN bus bit rate, in bit/s.
can.power_on		immediately	3, 5	0	1	1	Enable CAN bus power output.
can.terminator_on		immediately	5	0	1	1	Enable the embedded CAN bus termination resistor.
slcan.timestamping_on	Z	immediately	15, 16, 17	0	1	1	Time stamp all incoming and outgoing CAN frames.
slcan.flags_on		immediately	17	0	1	0	Enable flags for SLCAN notifications (SLCAN protocol extension).
uart.baudrate	U	immediately	7, 15	2400	3 000 000	115200	Baud rate of the UART port.

7 Embedded bootloader

7.1 Introduction

Babel employs the Zubax Embedded Bootloader – a highly robust open source bootloader designed for deeply embedded systems that can update firmware over USB and serial port. The bootloader also offers advanced integrity checking capabilities.

The bootloader starts immediately after the device has been powered on. Having started, the bootloader checks if there is a valid application (i.e. firmware) that can be executed. If there is one, the bootloader measures a 5 second timeout since the point of its initialization, and once the timeout has expired, the bootloader starts the application, unless an external entity has requested it to download a new application image or to wait longer. If there is no valid application found (i.e. nothing to boot), the bootloader will wait forever for commands.

The details about the supported communication interfaces are summarized in the table 7.1. Observe that this bootloader does not make use of the CAN bus interface. As long as the bootloader is running, the CAN controller remains inactive in order to avoid any interference with the CAN bus the device may be connected to.

The CAN power supply output and the CAN termination resistor remain inactive while the bootloader is running.

The bootloader is fully fault-tolerant. If the update process fails at any point for any reason (e.g. communication failure, power supply failure, and so on), the device may end up with a damaged application. The bootloader is able to recognize this condition and refuse to start the invalid application. In order to recover from this state, the update process simply needs to be restarted.

As an additional safety measure, the bootloader uses a hardware watchdog timer that allows it to abort applications that do not start properly. This minimizes the chances of permanently incapacitating the device¹⁵ by uploading a dysfunctional application image.

Table 7.1: Bootloader communication interfaces

Interface	Parameters	Protocol	Note
USB	CDC ACM (virtual serial port)	YMODEM, XMODEM, XMODEM-1K (autodetect)	When USB is connected, the UART interface is inactive.
UART	115200-8N1 (fixed)	Same as USB	Available only while USB is disconnected.

7.2 State machine

The behavior of the bootloader is defined by a simple state machine documented on the figure 7.1. The table 7.2 summarizes the states.

¹⁵Bricking.

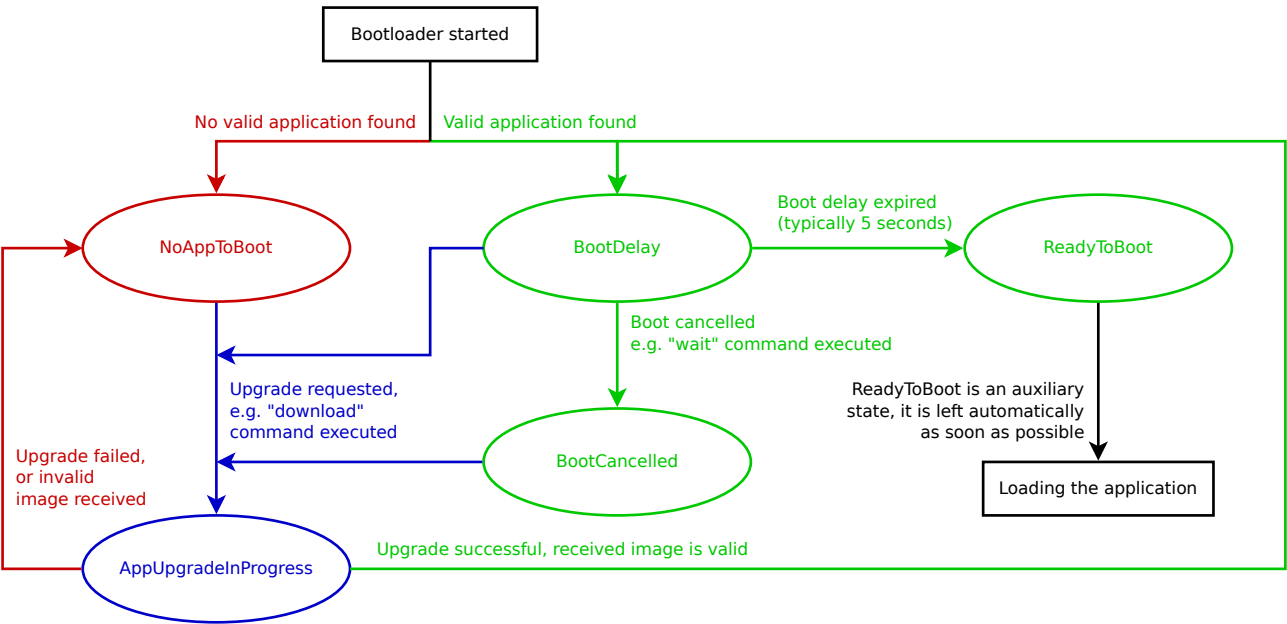



Figure 7.1: Bootloader state machine.

Table 7.2: Bootloader states

ID	Name	Description
0	NoAppToBoot	There is no valid application to boot; the bootloader will be waiting for commands forever.
1	BootDelay	The bootloader will start the application in a few seconds, unless booting is canceled or an application update is requested.
2	BootCancelled	There is a valid application to boot; however, booting was canceled by an external command.
3	AppUpgradeInProgress	The application is currently being updated. If interrupted, the bootloader will switch into NoAppToBoot or BootCancelled .
4	ReadyToBoot	The application is about to be booted. This state is very transient and is left automatically as soon as possible.

7.3 LED indication

While the bootloader is running, the LED indicators behave as described in this section.

The status LED (blue ) is always on, which is the main indicator that the bootloader, rather than the application (firmware), is currently running.




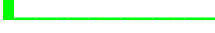

The CAN traffic LED (green ) displays one of the blinking patterns shown in the table 7.3, depending on which state the bootloader is in.

Table 7.3: Bootloader state indicated via the CAN traffic LED indicator

Bootloader state	LED pattern (step 50 ms)	LED behavior
NoAppToBoot		Blinking 10 Hz (very quickly)
BootDelay, ReadyToBoot		Turned off
BootCancelled		Blinking 1 Hz, short pulses (50 ms)
AppUpgradeInProgress		Blinking 1 Hz, long pulses (500 ms)

7.4 Error codes

The table 7.4 provides descriptions for the well defined error codes that can be reported by the bootloader.

Table 7.4: Bootloader error codes

Code	Description
0	Success.
1	Unknown error.
9001	Application ROM driver error: erase failed.
9002	Application ROM driver error: write failed.
10001	The current state of the bootloader does not permit the requested operation.
10002	Application image is too large for the device. Download has been aborted.
10003	Failed to write the next downloaded chunk of the application image into the ROM.
20001	X/YMODEM interface write has timed out.
20002	X/YMODEM retries exhausted.
20003	X/YMODEM protocol error.
20004	X/YMODEM transfer has been canceled by the remote.
20005	X/YMODEM remote has refused to provide the file.
32767	Unknown error.

7.5 Interface selection

Once started, the bootloader launches a CLI¹⁶ instance on the UART port. If the bootloader detects that the USB interface became active (by virtue of being connected to a USB host), it disconnects the CLI from UART, rendering the latter silent and unresponsive, and connects the same CLI instance to the USB virtual serial port. The CLI will remain available on the USB virtual serial port as long as the USB interface is active. Shall the USB port become disconnected, the bootloader will switch the CLI back to UART. The switching between USB and UART is fully automatic and happens on the fly.

7.6 USB interface properties

The USB interface will be detected by the host as CDC ACM (also known as virtual serial port). This is a standard USB class that is supported by vast majority of operating systems out of the box, no special drivers are required.

The bootloader reports the following properties to the USB host:

- Vendor ID – 0x1D50
- Product ID – 0x60C7
- Vendor string – Zubax Robotics
- Device description string – Zubax Babel Bootloader
- Device ID – the 128-bit globally unique device ID (section 2.5) as a hexadecimal string

7.7 UART interface properties

The UART interface has the following properties that cannot be changed:

¹⁶Command line interface.

- Baud rate – 115200
- Word size – 8 bit
- Parity control – none
- Stop bits – 1

7.8 CLI properties

The CLI uses the CR-LF line ending sequence (`\r\n`). In order to avoid unintended side effects of exposing the bootloader's CLI when the outer hardware expects an SLCAN interface, the bootloader's CLI does not return any echo until the input of a valid command has been completed.

For example, entering a command `zubax_id` will not produce any echo, until the command has been completed with the `\r\n` sequence. As such, entering `zubax_id\r\n` will return the echo followed by the command output. Echo for invalid commands is never returned.

The delayed echo allows the bootloader to silently ignore all SLCAN traffic and other data it doesn't recognize. Otherwise, the echo emitted by the bootloader could be misinterpreted by the connected hardware as valid SLCAN data.

Due to the above considerations, the CLI does not offer any prompt.

7.9 CLI commands

This section documents the CLI commands that can be of interest to the end user. Some commands that are not intended for use in production are intentionally omitted from this reference.

7.9.1 **reboot**

Restarts the bootloader normally.

7.9.2 **wait**

Instructs the bootloader to not boot the application automatically.

If the current state is `BootDelay`, the state will be switched to `BootCancelled`. In all other states the command will have no effect.

7.9.3 **download**

Instructs the bootloader to start an YMODEM/XMODEM/XMODEM-1K receiver on the current serial link and await for the remote host to begin transmission of the new application image file.

The bootloader will automatically detect which file transfer protocol to use.

According to the YMODEM specification, if no transfer was initiated by the host within one minute, the command will exit with an error. Possible error codes are defined in the table 7.4.

Note that while this command is running, the CLI will be unavailable, because the same serial link will be temporarily occupied by the file transfer protocol. Automatic switching between USB and UART is not available while the command is running.

See the section 7.10 for the detailed information about the implementation.

7.9.4 **zubax_id**

This is a standard command documented in the section 4.4.1.2. Its implementation in the bootloader, however, has a number of additional features.

The software version information provided in the output is obtained from the application that is currently installed on the device. If the bootloader could not find any installed application, the software version fields will be omitted from the output.¹⁷

The version of the bootloader itself is reported via the following set of dedicated fields:

- `bl_version` – bootloader version, major and minor, formatted as `<major>.<minor>`
- `bl_vcs_commit` – bootloader version control system commit identifier as an integer number.
- `bl_build_date` – the build date of the bootloader.

An additional field named “mode” is set to the string “bootloader” to indicate that the bootloader is currently running rather than the application.

The table 7.5 summarizes the fields reported by the bootloader. Some extra fields may be reported as well, which are not documented here because they are not designed for production use.

Table 7.5: Zubax ID fields

Field name	Meaning
<code>product_id</code>	Product type identifier string. The same string is reported via UAVCAN as the node name string.
<code>product_name</code>	Human-readable product name.
<code>mode</code>	Set to the string “bootloader” to indicate that the bootloader is running.
<code>sw_version</code>	Application version number in the form “major.minor”. Omitted if the application could not be found.
<code>sw_vcs_commit</code>	Application version control system commit identifier. Omitted if the application could not be found.
<code>hw_version</code>	Hardware version number in the form “major.minor”.
<code>hw_unique_id</code>	The 128-bit unique ID of this specific hardware instance (section 2.5).
<code>hw_signature</code>	The certificate of authenticity (CoA) of this specific hardware instance encoded in a Base64 string. If this data is missing, please inform Zubax Robotics as soon as possible.
<code>bl_version</code>	Bootloader version number in the form “major.minor”.
<code>bl_vcs_commit</code>	Bootloader version control system commit identifier.

7.9.5 state

Prints the current state of the bootloader in the following form:

```
| StateName (StateID)
```

Where `StateName` is the name of the current state as specified in the table 7.2, and `StateID` is the numerical identifier of the current state.

An example output is shown below:

```
| BootCancelled (2)
```

7.10 YMODEM/XMODEM/XMODEM-1K implementation details

YMODEM, XMODEM, and XMODEM-1K are simple and popular file transfer protocols designed by Ward Christensen and Chuck Forsberg. You can learn more about these protocols in the Zubax

¹⁷Version 1.0 of the bootloader used to employ a different convention where the application version fields were using a different prefix: `fw_` rather than `sw_`.

Knowledge Base at <https://kb.zubax.com/x/ZwAz>. This section elaborates on the noteworthy implementation details specific to this application.

The `download` command starts a multi-protocol receiver. The receiver enters a loop where it emits the ASCII NAK character to the host, prompting it to begin transmission of the application image file. The receiver will emit NAK every 5 seconds until the host begins the transmission, until the transfer initialization times out, or until the host cancels the transmission, whichever happens first. The transfer initialization timeout is set to 1 minute.

The receiver always uses the NAK character to initiate transfers rather than “c”, which instructs the remote host to use the plain 8-bit checksum for data integrity checking rather than CRC-16. The data integrity guarantees offered by the plain 8-bit checksum algorithm are deemed sufficient, because the data links are considered reliable enough, and the application image itself is always protected by a strong CRC function.

Being compatible with three different protocols, the receiver supports the following options:

- The host is free to send the zero block with the file metadata, as defined by YMODEM. The receiver will collect the file size information from the metadata packet and ignore the rest.
- The host is free to use either 256-byte or 1024-byte sized blocks, the receiver supports both. The former are defined by YMODEM and XMODEM, the latter are defined by YMODEM and XMODEM-1K.

Note that if the size of the application image file has not been provided, the written image will be padded up to the size of the last data block. This is acceptable, because the trailing bytes after the application image are not used by any part of the system, and as such their contents can be arbitrary. It is recommended, however, to fill the padding bytes with 0xFF, in order to match the initial state of the ROM.

There is a large number of software products and scripts that support these file transfer protocols. For instance, the popular program `sz` (available on most GNU/Linux distributions) can be used as follows (where `$file` is the name of the application image file, and `$port` is the name of the serial port):

```
| sz -vv --yodem --1k $file > $port < $port
```

There are various GUI-based alternatives for Windows and Mac OS as well.

A Appendix: Third party SLCAN API implementations

SLCAN TTY Line Discipline

API Comparison of existing serial CAN Protocol Adapters

SLCAN	Function	CAN232	CANUSB	Micronics	CANhack	Remarks
	green = common functions	www.can232.com	www.canusb.com	www.micronics.de	www.canhack.de	
	End of Command	lr (CR)	lr (CR)	lr (CR)	lr (CR)	Computer -> CAN Adapter
	ACK	lr (CR)	lr (CR)	lr (CR)	lr (CR)	CAN Adapter -> Computer
	NACK	la (BEL)	la (BEL)	la (BEL)	la (BEL)	CAN Adapter -> Computer
	Setup Btrrate	Sx (0 <= x <= 8)	Sx (0 <= x <= 8)	Sx (0 <= x <= 8)	Sx (1 <= x <= 8)	no support for S0 (10kBit/s)
	Setup BTR	sxxxx (xxxx = hex)	sxxxx (xxxx = hex)	sxxxx (xxxx = hex)	sxxxxx (xxxxxx = hex)	no SJA1000 => 6 BTR digits
	Open Channel	O	O	O	O	Operation Mode
	Open in Listen Only Mode			L	L	Listen Only Operation Mode
	Close Channel	C	C	C	C	Reset Mode
	TX/RX Frame Format SFF	tiilddddd...	tiilddddd...	tiilddddd...	tiilddddd...	0 <= 'l' <= 8
	TX/RX Frame Format EFF	Tiiiiilddddd...	Tiiiiilddddd...	Tiiiiilddddd...	Tiiiiilddddd...	0 <= 'l' <= 8
	TX/RX Frame Format RTR/SFF	riiil	riiil	riiil	riiil	undocumented but in demo
	TX/RX Frame Format RTR/EFF	Riiiiiii	Riiiiiii	Riiiiiii	Riiiiiii	source code. 'l' should be 0
						RTR support since firmware v1.20
	Poll single frame	P				
	Poll all frames in FIFO	A				
	Auto Poll	Xx (0 <= x <= 1)				written in EEPROM
	UART Speed Setup	Ux (0 <= x <= 6)				written in EEPROM
	Read Arbitration Lost Register			A		
	Read Error Capture Register			E		
	Read SJA1000 Register			Gxx		
	Write SJA1000 Register			Wrrdd		
	Read Status Flags	F	F	F	F	With FIFO Information
	Timestamp On/Off	Zx (0 <= x <= 1)	Zx (0 <= x <= 1)	Zx (0 <= x <= 1)	Zx (0 <= x <= 1)	written in EEPROM
	Acceptance Mask	Mxxxxxxx	Mxxxxxxx	Mxxxxxxx	Mxxxxxxx	
	Acceptance Value	mxxxxxxx	mxxxxxxx	mxxxxxxx	mxxxxxxx	
	HW/SW Version	V	V	V		Vcccclr
	Major/Minor Version			v	v	vcccclr
	Serial Number	N	N	N		Ncccclr

<http://developer.berlios.de/projects/socketcan>

(c) 2008 Oliver Hartkopp

Figure A.1: An overview of the SLCAN API implemented in various SLCAN-compatible adapters.

Prepared by Oliver Hartkopp (Linux SocketCAN project).