DI-Net Software Design Specification

Software Design Specification

Verification	Date:	Validation	Date:

1. General information

This software design document describes the DI-Net CAN protocol, software and firmware library components architecture. Which are the building blocks for all the devices which communicate over DI-Net.

1.1 Changelog

Version	Date	Changes

1.00	09-11-2016	• Migrate generic DI-Net documentation from TWS-3000 Firmware design specification.
		• Change DI-Net CAN protocol ABI
		  ○ Specify message dictionary
		  ○ Add network maintenance messages
		  ○ Add reply error messages
		  ○ Add source and destination node ids fields
		  ○ Add message types and message size fields
		• Libdi_fw add UI, heartbeat, charger and battery measurement components
		• Update EEPROM driver
		• Add DNCM chapter
		• CAN: Add new messagetype log
		• Libdi: Add logging facility
		• DNCM: Add sdcard logging
		• Fix battery indicator critical and empty percentage to 5%
		• Change heartbeat interval from device "armed" to correct "active" state
		• Fix diagram dimensions in Section 7.3

1.01	24-11-2016	• Add DI-Net CAN node redundancy • Add DNCM redundancy
1.02	01-12-2016	• Add section DI-Net CAN-bus node redundancy
1.03	03-04-2017	• Add CAN NET node message to dictionary for error publish • Add device "`service`" state • Add DNCM DI-Net RPC `device:uid` section
1.04	31-05-2017	• Add Xbee chapter to DNCM
1.05	27-07-2017	• Add CAN DNCM raw commands to control gps sensor data trigger, communication status reporting • Add DNCM can device module documentation

1.06	24-11-2017	• Add di firmware library config self test and factory defaults sections
1.07	12-03-2018	• CAN: Add battery publish raw message for multi-node device (TWS-73) • Charger & battery, document status reporting behaviour based when connected and time is synchronized (TWS-91) • Update references to latest releases
1.08	14-03-2018	• Process review comments of v1.07 • Add battery indicator section

1.09	08-05-2018	• Process review comments of v1.08
		• DNCM
		  ○ Update architecture diagram based on v4 hardware (DINET-41)
		  ○ Add MCP8908 temperature sensor driver (DINET-41)

1.10	19-06-2018	• DNCM
		  ○ Split temperature reporting into its own chapter (previously part of MCP8908 driver) (DINET-41)
		  ○ Update GPS according to hardware v4 (DINET-40)
		• CAN
		  ○ Change can message DNCM_GPS_SENSOR_TRIGGER payload type from U32 to correct U16 (DINET-40)

1.11	xx-xx-2018	• DNCM: Raise leader conflict error and report to server (DINET-10) • CAN message dictionary: Add explicit note about DI_CAN_NET_DTYPE_NODE_UID may only be a unicast request (DINET-68) • CAN message dictionary: Document RPC messages (DINET-69)
1.12	13-09-2018	• CAN message dictionary: add raw CAN req-rep messages for fetching HW and FW versions from other nodes using the CAN-bus. • CAN message dictionary: remove NODE_FW_VERSION section (now replaced by RAW_FW_VERSION and RAW_HW_VERSION)

1.13	31-10-2018	• CAN message dictionary: stringlength for replying firmware versions is now 60 (was: 40) • Redefine Transaction ID
1.14	14-01-2019	• CAN message dictionary: add ZKL_GET_TEMPERATURE raw message type • Introduce charging delay to avoid charging the battery too often • charger: Introduce method to determine if the battery is not present or empty (both result in 0V)
1.15	19-02-2019	• CAN message dictionary: add RWM_SENSOR_DATA_READY raw message type • Add DI-Net RPC Realtime send over UDP protocol (TWS-190) • Add DI-CAN RT message flag and sequence number to support the UDP protocol for Realtime messages (TWS-192)
1.16	25-02-2019	• Fix review comments from rlieshout • Add DI-CAN emflags compatibility with older firmware so no all messages are routed using RT flag (TWS-192) • Add DNCM DI-CAN RT message routing (TWS-192)

1.2 Review

REV = reviewer, VER = verifier, VAL = validator, AUT = author

Name	1.00	1.01	1.02	1.03	1.04	1.05
ing. J.J.J. Jacobs	AUT	AUT	AUT	AUT	AUT	AUT
ir. R.J.W.Habets	VER					
ing. L.J.M. van der Poel	VAL					
ir. R.H. Lieshout		REV	REV	REV		REV
ing. A.A.W.M. Ruijs			REV		REV	
ing. R.W.A. van der Heijden				REV	REV	
S.F.S van der Vlies					AUT	

Name	1.06	1.07	1.08	1.09	1.10	1.11
ing. J.J.J. Jacobs	AUT	AUT	AUT	AUT	AUT	AUT
ir. R.H. Lieshout	REV	AUT	AUT	AUT	REV	

Name	1.12	1.13	1.14	1.15	1.16	
ing. J.J.J. Jacobs	REV	REV	REV	AUT	AUT	
ir. R.H. Lieshout	AUT	AUT	AUT	AUT	REV	
ing. A.A.W.M. Ruijs			REV			

1.3 Appendices

[1]	Title:	Linux kernel coding style
	Author (Company):	Linux Kernel Community
	Date:	2017-10-19
	File/URL:	https://www.kernel.org/doc/html/v4.10/process/coding-style.html
[2]	Title:	Specification Document of DI-Net Remote Procedure Protocol (RPC)
	Author:	Dual Inventive
	Version, Date:	v3.2.0, 2018-02-13
	File/URL:	71023 - DI-Net Remote Procedure Protocol v3.2.0.pdf
[3]	Title:	DI-Net Software Design Specification DI-Net CAN Messages Dictionary
	Author:	ing. J.J.J Jacobs, ir. R.H. Lieshout
	Version, Date:	v1.16, 2019-02-25
	File/URL:	71023 - DI-Net Software Design Specification CAN Messages Dictionary - 1.16.pdf
[4]	Title:	MTinfo 3000 Backend Design Specification
	Author:	ing. R.W.A. van der Heijden, ing. J.J.J. Jacobs, ing. J.C.M. Raats
	Version, Date:	v3.21, 2018-04-19
	File/URL:	71020 - Backend Design Specification v3.21.pdf
[5]	Title:	AirPrime HL6 and HL8 Series AT Commands Interface Guide
	Author:	Sierra Wireless
	Version, Date:	January 18, 2016. Rev 12
[6]	Title:	Raft Consensus Algorithm
	Author:	Diego Ongaro and John Ousterhout.
	File/URL:	https://raft.github.io [Last accessed 24 November 2016]
[7]	Title:	AirPrime HL76xx AT Commands Interface Guide
	Author:	Sierra Wireless
	Version, Date:	December 17, 2018. Rev 11.0

1.4 Markings and abbreviations

1.4.1 Markings

Marked text	Text needs to be changed or completed
Marked text	Text has changed compared to the previous version
Marked section	Section for review

1.4.2 Abbreviations

Abbreviation	Description
CAN	Controller Area Network

CRC	Cyclic Redundancy Check
DI	Dual Inventive
DI-Net	Dual Inventive Network
DNCM	DI-Net Communication Module
EEPROM	Electrically Erasable Programmable Read-Only Memory
FW	Firmware
GC	Garbage Collector/Collection
HAL	Hardware Abstraction Layer
I^2C	Inter-Integrated Circuit
ID	Identifier
LED	Light Emitting Diode
LFSR	Linear Feedback Shift Register
MCU	Microcontroller Unit
MFT	Multi Frame Transfer
PRNG	Pseudo Random Number Generator
Pub	Publish
Rep	Reply
Req	Request
RPC	Remote Procedure Call
RSSI	Received Signal Strength Indicator
RT	Realtime
RTOS	Real-time Operating System
SDIO	SD Input/Output
SFT	Single Frame Transfer
SPI	Serial Peripheral Interface
Sub	Subscribe
TCP	Transmission Control Protocol
UART	Universal Asynchronous Receiver/Transmitter
UDP	User Datagram Protocol
UID	Unique identifier

Table of contents

2. Introduction

2.1 Coding style

The used coding style within all DI-Net firmware uses Linux kernel coding style (See ref[1]). Major benefit is the availability of a tool which can analyze and report style errors. The tool "`checkpatch.pl`" is used from the Linux kernel repository.

A few deviations are performed from this style to make conforming to this style easier. The changes are:

- Maximum line-length is set to 120 characters
- Initialization of statics to `` `0` `` or `NULL` is allowed, it is not necessary within the C programming language, but it is more implicit in firmware.
- File-scoped (static) variables are prefixed with `` `g_` ``

2.2 Version control

All code is stored in the Dual Inventive Git repositories. The software and firmware is spread along different components.

Repository	Components & description
dinet/libdi.git	DI-Net common C library
dinet/libdi-php.git	DI-Net PHP extension, with DI-Net CAN support (based on libdi)
dinet/libdi_fw.git	DI-Net common firmware C library
dinet/fw-ChibiOS.git	ChibiOS RTOS
fw/dncm.git	DNCM firmware

3. ChibiOS/RT RTOS

A real-time operating system is chosen to reduce code complexity and separate functionality into different tasks. ChibiOS/RT is chosen because of the following points:

- Full support for STM32 microcontrollers
- Abstracts Low-Level hardware interface (HAL) (for improved portability to other hardware platforms)
- MISRA-C 2012 compliant (= automotive standard)
- Ultra-fast (222.000 thread-creations/deletions and 1.2 µs context switch @72MHz on STM32 processor)
- Open-source & commercial with support
- Extensive documentation and examples
- Large user-base, mature implementation (since ~2006)
- Increased system reliability by a complete static kernel architecture
- Fair licensing

ChibiOS/RT is the kernel and ChibiOS/HAL are the drivers, further in this document it will be referenced as ChibiOS. A general overview of the firmware with ChibiOS is drawn in Figure 1.

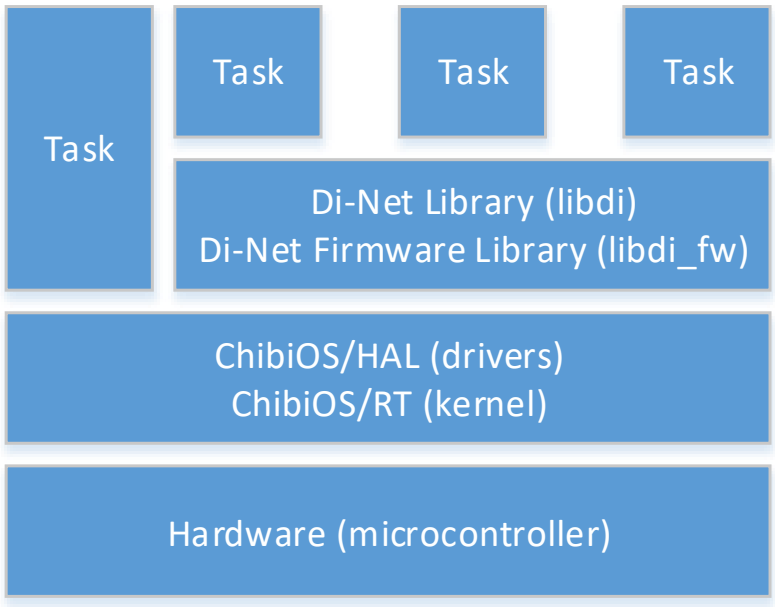


Figure 1 Firmware overview with ChibiOS/RT

4. DI-Net common C library (`libdi`)

The DI-Net framework is a common function library written in cross-platform C11.

4.1 Features

- End-to-end encryption
- Timekeeping and synchronisation
  - System time (date and time)
  - Measurement/monitoring time, millisecond granularity
- DI-Net CAN Bus Protocol stack
  - Node redundancy
- Logging facility

4.2 End-to-end encryption

The security layer interfaces with the TCP-layer and implements the CCKE-PSK-AES-128-CBC-HMAC-MD5-IVFI algorithm for communication with DI-Net devices as described in ref[4] - 3.1.3 Security/Encryption.

4.3 Timekeeping and synchronisation (`di_time`)

Time synchronisation is necessary for timestamping of measurements with an absolute time. Relative time (since boot) is used for general timekeeping e.g. for timeouts, status reporting intervals etcetera.

UNIX timestamp epoch/zero with millisecond granularity:

Start value: 1 January 1970 00:00:00.000 UTC

An absolute time is expected to be valid if it is at least 1 January 2016 00:00:00.000 UTC.

4.3.1 GPS date-time to timestamp

The following tasks are performed to calculate a timestamp from GPS time:

- Read GPS date-time from modem (e.g 2015-08-10 10:31:22)
- Calculate day of year, which is necessary for UNIX timestamp calculation
  - Calculate for every month until now() the amount of days
    - Calculate amount of days for month (keeping leap years in account)
- Calculate UNIX timestamp (see 4.3.2)

4.3.2 UNIX timestamp calculation

Calculation of the UNIX timestamp is performed in the following way:

> *A value that approximates the number of seconds that have elapsed since the Epoch. A Coordinated Universal Time name (specified in terms of seconds (tm_sec), minutes (tm_min), hours (tm_hour), days since January 1 of the year (tm_yday), and calendar year minus 1900 (tm_year)) is related to a time represented as seconds since the Epoch, according to the expression below.*

If the year is <1970 or the value is negative, the relationship is undefined. If the year is >=1970 and the value is non-negative, the value is related to a Coordinated Universal Time name according to the C-language expression, where tm_sec, tm_min, tm_hour, tm_yday, and tm_year are all integer types:

```
tm_sec + tm_min*60 + tm_hour*3600 + tm_yday*86400 +
    (tm_year-70)*31536000 + ((tm_year-69)/4)*86400 -
    ((tm_year-1)/100)*86400 + ((tm_year+299)/400)*86400
```

The relationship between the actual time of day and the current value for seconds since the Epoch is unspecified.

How any changes to the value of seconds since the Epoch are made to align to a desired relationship with the current actual time is implementation-defined. As represented in seconds since the Epoch, each and every day shall be accounted for by exactly 86400 seconds.

Note:

The last three terms of the expression add in a day for each year that follows a leap year starting with the first leap year since the Epoch. The first term adds a day every 4 years starting in 1973, the second subtracts a day back out every 100 years starting in 2001, and the third adds a day back in every 400 years starting in 2001. The divisions in the formula are integer divisions; that is, the remainder is discarded leaving only the integer quotient.

Original source:
http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_15

4.3.3 TCP server timestamp

The TCP server timestamp is directly read and no further calculation is necessary (See ref[2] – 32. Low-level protocol).

4.3.4 Distributed time synchronisation

Distributed time synchronisation from the DNCM to the "Slave CPUs" is performed according to the architecture drawn in Figure 2. The CAN-bus is used to synchronize with the "Global GPS time" and the local (one millisecond) timer is used to update the local time with one millisecond tick.

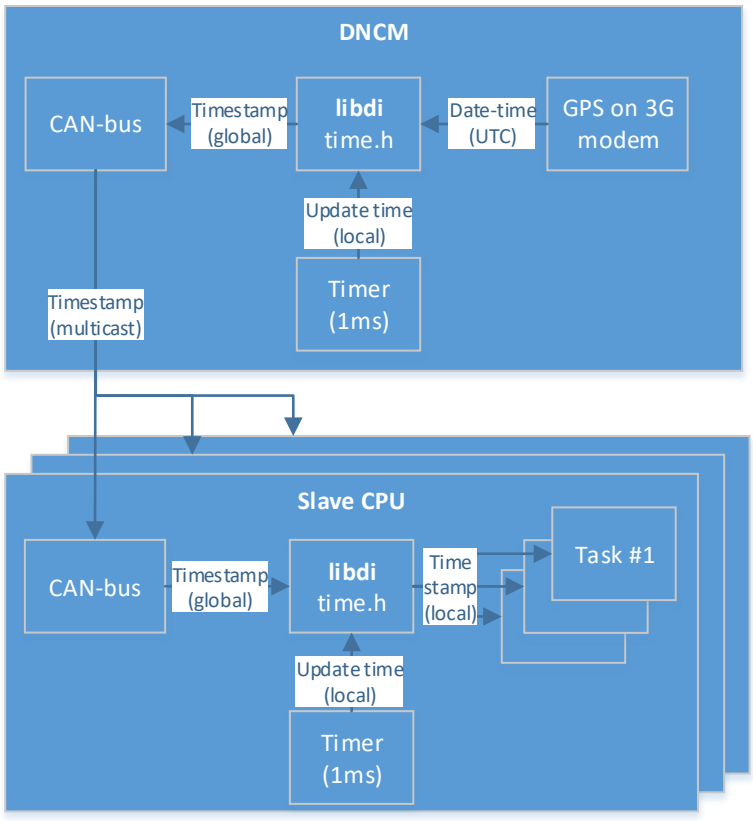


Figure 2 Distributed time synchronization overview

4.4 Device-states with Service and Lock/Unlock

When the business logic locks and unlocks the device a global device state is modified. The states are generic and state-transition actions are device specific (e.g calibration, alarming, visual appearance). The lock and unlock states are non-volatile and are saved in the device EEPROM. The states are composed of two configurable variables. Labeled in the RPC config class as "token", "activate" and "service" (See ref[2]). The device states and legal transitions are depicted in Figure 3.

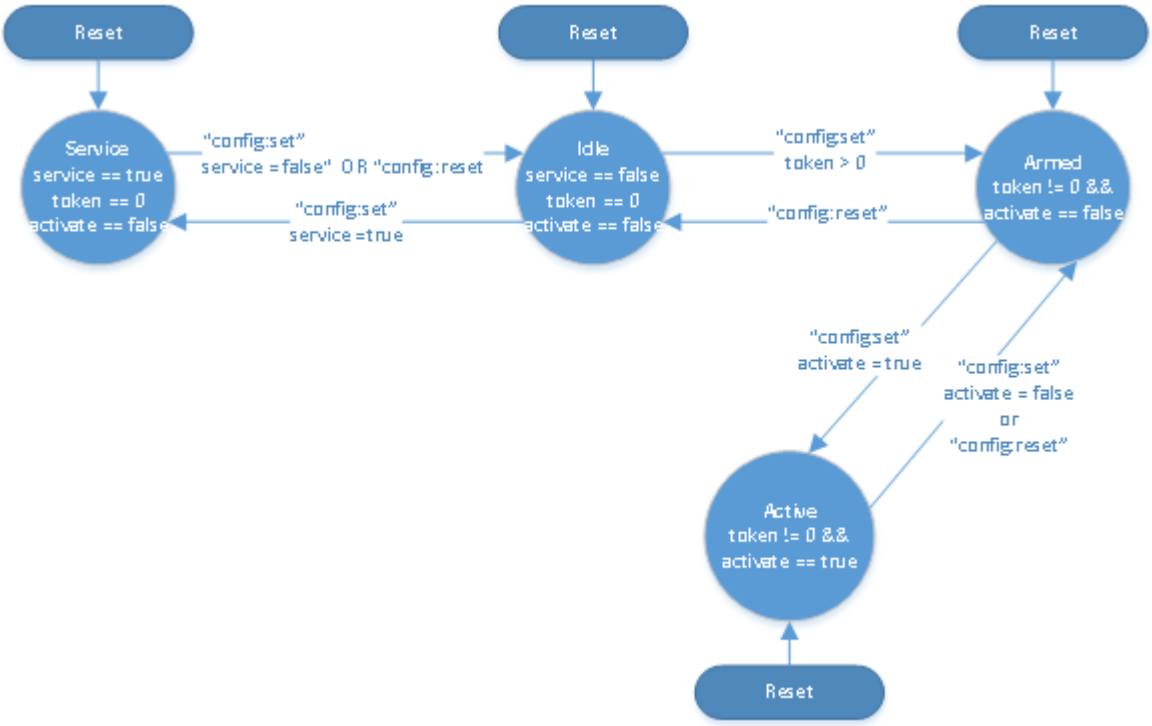


Figure 3 Device Lock and Unlock states

- In the device "`active`", "`armed`", "`idle`" state the RPC message "`device:ping`" is published with an interval of 60 seconds.
- In the device "`active`" state the RPC message "`device:ping`" is published with an interval of one second.

Error checking

When RPC "`config:set`" is executed the message value type is checked, when it doesn't match as expected a di_errno `` `DNE_OPDENIED` `` is returned to the requester. It is possible the requester tries to execute an invalid transition then `` `DNE_OPDENIED` `` is returned.

4.5 Generic Non-volatile configuration API (`di_config`)

For non-volatile device and chip configuration a generic API is created. It provides a high level interface for the application to load and store configuration data in a platform independent way. The generic API has a modular low-level interface to support different storage drivers.

The application defines a list of configuration items with a unique id and a size in bytes. The configuration API takes care of per-configuration item integrity with the CRC-CCIT16 algorithm.

On configuration set (write) operation the API checks if the write size is less or equal the item can hold. Then the CRC is calculated and a low-level write for CRC is performed. Then

a next write is performed for the application data. The low-level driver may supply a synchronize call-back. The `sync` action is used to flush flash page caches. And is used when a Read-Modify-Write driver is necessary to write flash memory.

The configuration get (read) operation performs the opposite. It first reads the CRC from the low-level layer, then it reads the data into the application buffer and then checks if it matches.

The low-level driver may locate the configuration items where necessary. But must keep them in sequential order.
In Figure 4 the architecture of the generic configuration API is drawn.

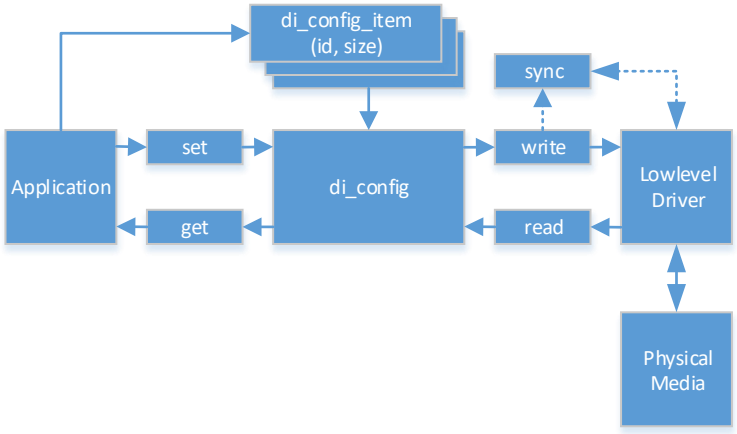


Figure 4 libdi generic configuration API overview (`di_config`)

4.6 Logging facility (`di_log`)

The logging facility is used to have runtime tracing of events. This is a generic module which can log to different streams (stdio, uart, can-bus). A logging writer must be initialized before the application is able to log. The logging facility is thread safe.

A single log message is composed as follows:

- timestamp (`uint64_t`): DI-Net timestamp (unix time with millisecond granularity)
- loglevel (`enum di_log_level`)
  - CRIT: a critical error message
  - ERROR: an error message
  - WARN: a warning message
  - INFO: informational message
  - DEBUG: debug message

- component (`enum di_log_component`): the component where the message is generated (libdi, libdi_fw, application firmware)
- module: the module where the message is generated (cloudlight, tcp, modem) the module is based on the component (e.g component libdi -> module device)
- msg: the human readable log message string

The library or application only needs to call the `` `DI_LOG(loglevel, const char *fmt, ...)` `` function macro.

The component and module identifiers are derived from the following macros:

- `DI_LOG_COMPONENT`: must be set project-wide
  - With compiler flag: `` `-DDI_LOG_COMPONENT=DI_LOG_COMPONENT_LIBDI` ``
- `DI_LOG_MODULE`: must be set per-file/module before including `<di/log.h>`
  - `#define DI_LOG_MODULE DI_LOG_MODULE_DI_FW_CLOUDLIGHT`

The logging facility can be configured at compile time with the following macros:

- When `DI_LOG_QUIET` is defined the logging is completely disabled
- When `DI_LOG_ENABLE_FROM_<loglevel>` is defined only the log messages are compiled-in starting from the enabled loglevel. E.g: when `DI_LOG_ENABLE_FROM_WARN` is defined the CRIT, ERROR, WARN are only compiled-in. And INFO, DEBUG is removed.

4.7 DI-Net CAN-bus node redundancy

The DI-Net CAN-bus node redundancy algorithm is derived from the Raft Consensus Algorithm (see ref).

The minor difference between DI-Net CAN-bus redundancy and Raft is that uses a timeout-based leader-follower takeover algorithm instead of majority voting. The rationale is because there is only a requirement of two redundant nodes (who are of the same type) and reduces software complexity.

Log (data) replication is not in the scope of the DI-Net CAN-bus node redundancy. As no nodes replicate between each other.

4.7.1 Node roles

All nodes on the DI-Net CAN-bus fulfil a role (only one role at a given time).

Passive

Passive nodes can function on their own and are not part of a redundant configuration. A node can also be passive when it is booting/initializing and has is not ready yet to fulfil a follower/leader role.

Follower

Follower nodes monitor broadcasted heartbeats from other leaders (of the same node type). Follower nodes are "active" and fulfil partial (to reduce power consumption) or all

functionality (due to hardware restrictions). A follower "automatically" becomes leader when the monitored leader heartbeat timeout is exceeded.

Leader

Leader nodes broadcasts periodically a heartbeat on the bus. Leaders are "active" and fulfil all the node functionality.

5. DI-Net common firmware C library (`libdi_fw`)

In this chapter architecture and design considerations are described for the DI-Net firmware library (`libdi_fw`). The library is used for common functionality in the device firmware.

5.1 Configuration low-level EEPROM driver (Microchip 24xx)

In this section the low-level storage configuration driver is described, see section 4.5 for the generic configuration API. This driver is used with Microchip 24xx I2C EEPROMs.

The driver hides the data paging from the application. Paging is only necessary on write operations. While read operations are supported randomly at byte-level.

The low-level I2C driver is not depicted in this section because it uses the ChibiOS HAL, see chapter 3 for full-context.

On write the driver calculates how many pages need to be written. The page size of the EEPROMs is always 8 bytes. The configuration API supplies the offset and size it want to write into. We need to translate this in the following way:

- Calculate current page
- Calculate position on current page
- Calculate remaining bytes for the current page
- Write the current page bytes
- Write full pages
- Write remaining bytes

$$PAGE_{SIZE} = 8$$

$$PAGE_{current} = \frac{offset}{PAGE_{SIZE}}$$

$$PAGE_{position} = offset \,\%\, PAGE_{SIZE}$$

$$Remaining_{firstpage} = PAGE_{SIZE} - PAGE_{position}$$

$$pages = \frac{(size - Remaining_{firstpage})}{PAGE_{SIZE}}$$

$$Remaining_{lastpage} = (size - Remaining_{firstpage}) \,\%\, PAGE_{SIZE}$$

In case the size Remaining first page is bigger than the size given by the application only that page will be written and no further calculations are needed.

5.2 User interface

The user interface is made up from 3 separate components; the CloudLight, the Battery indicator and the UI-button.

5.2.1 CloudLight (`di_fw_cloudlight`)

The CloudLight is a UI element to show the combined status of:

- DNCM TCP connection state
- Device or system error(s)

All possible states with corresponding appearances are described in Table 1.

Table 1 CloudLight states

Color	Appearance	Connection	Errors	State
Blue	Fading 0,5 hertz	No	No	"offline"
Blue	Solid	Yes	No	"online" *
Red	Solid	Yes	Yes	"online-errors"
Red	Fading 0,5 hertz	No	Yes	"offline-errors"
n/a	n/a	Yes	No	"powersave" *

* The "online" state is shown for 20 seconds and will transition automatically into the "powersave" state, which disables all LEDs. The user is able to press the UI-button to show the "online" state for another 20 seconds (see 5.2.3).

5.2.2 Battery indicator (`di_fw_battery_indicator`)

The battery indicator shows the current battery level and will indicate an active charger. The time the LEDs are visible is combined with the CloudLight. If the CloudLight is in power save so is the battery indicator. The battery indicator lights have 5 active states. In case the charger is not active, the indicator will show the following patterns:

Table 2 Battery indicator states

State	Color and appearance	Range
"full"	3 leds, Green (3, 2, 1)	> 66%
"half"	2 leds, Green (2, 1)	> 33% and ≤ 66%
"low"	1 led, Green (1)	> 15% and ≤ 33%
"critical"	1 led, Red	> 3% and ≤ 15%
"empty/removed"	none	≤ 3%

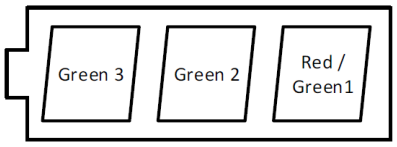


Figure 5: battery indicator

When the charger is active, a blinking LED will indicate the charger activity. The following patterns are used in that case:

Table 3 Battery indicator states when charging

State	Color and appearance
"full"	Green 2 and 1 continuous, Green 3 blinking 0,5 Hz
"half"	Green 1 continuous, Green 2 blinking 0,5 Hz
"low"	Green 1 blinking 0,5 Hz
"critical"	Red blinking 0,5 Hz
"empty/removed"	Red blinking 0,5 Hz

The battery indicator shows the battery and charger status when the device is ON or OFF. In the ON state, the indicator will behave according to the UI. In the OFF state, the battery indicator is only active when the charger is active. If not, the battery indicator will be OFF.

5.2.3 UI-button (`di_fw_cloudlight`)

The UI-button is used to show the device status to the user using the CloudLight and Battery indicator. When the Cloudlight is in the "powersave" state and the button is pressed the "online" state is shown for 20 seconds and will automatically transition into the "powersave" state.

5.3 Heartbeat (`di_fw_heartbeat`)

The heartbeat is a periodically published `device:data` RPC message.

- Send every 60 seconds when device state is `idle` or `armed`
- Send every second when device state is `active`

5.4 Battery measurement

The battery measurement is done by an ADC on the microprocessor. The calculation for this is depicted in section 5.4.1. This calculation is done for each battery in the device. After the calculation is done the state is determinate by set up values. This information will be send periodically (ever 15min) to the server trough RPC messages. If there is a state change or a voltage change of more than 0.1V it will be send over earlier. If the ADC conversion fails it retries after set interval and it sets an error on the device till next good conversion.

5.4.1 Calculation

To determinate the battery voltage out of the ADC measurement we firs need to know the volt per bit, to get this value we need the next formula:

$$ref\ volt = 3.3$$

$$data\ steps = 4096$$

$$noice\ reduction\ value = 8$$

$$volt\ per\ bit = \frac{ref\ volt}{(\frac{data\ steps}{noice\ reduction})} = 0.0064453125$$

After we know the volts per bit we now can use that to calculate the battery voltage in accordance with the ADC value.

$$resistor\ scaling\ value = \frac{75k + 10k}{10k} = 8.5$$

$$\frac{adc\ value}{noice\ reduction} * volt\ per\ bit * resistor\ scaling\ value = bat\ voltage$$

5.4.2 Battery status

The battery voltage is converted to a battery status. There are 7 levels as Table 4 shows.

Table 4 battery states

#	state	description
0	DI_DEVICE_BATTERY_STATE_REMOVED	Battery not connected
1	DI_DEVICE_BATTERY_STATE_EMPTY	Empty
2	DI_DEVICE_BATTERY_STATE_CRITICAL	Critical
3	DI_DEVICE_BATTERY_STATE_LOW	Low
4	DI_DEVICE_BATTERY_STATE_HALF	Half full
5	DI_DEVICE_BATTERY_STATE_FULL	Full
6	DI_DEVICE_BATTERY_STATE_UNKNOWN	Unknown

Each level corresponds to an upper- and lower limit voltage. Since these limits are determined by the type of battery that is present in the device, the limits are not defined in this library (`libdi_fw`) but are provided by the application specific `board.h` header file. Table 5 shows the voltage limits that need to be defined in the board header file in the application.

Table 5 battery voltage limits

Enumeration	Description
DI_FW_BATTERY_VALUE_66_PCT	>= 66% voltage value
DI_FW_BATTERY_VALUE_33_PCT	>= 33% and < 66% voltage value
DI_FW_BATTERY_VALUE_15_PCT	>= 15% and < 33% voltage value
DI_FW_BATTERY_VALUE_03_PCT	>= 3% and < 15% voltage value
DI_FW_BATTERY_VALUE_00_PCT	< 3% voltage value

5.4.3 Status reporting

Both the battery voltage as well as the battery status get published to the server using an RPC message. As soon as the device is connected (connection status reported on the CAN-bus by a gateway e.g DNCM), and the time is synchronized it starts to publish messages. This way the device never publishes data immediate on powerup.

tws-3000-wum device type specific behaviour

The battery level is also published over the local CAN bus. This way, a setup that consists of more than 1 board (like a WUM), is informed about the battery status of the other board(s).

5.5 Battery Charger

In case an external power supply is connected, the device is capable of charging the internal battery. The charger component will continuously monitor the charger input voltage (external power supply). Charging only takes place if the charging voltage is a valid range: [6..30Volt].

The charger can be in any of the following states:

State	Description	Voltage
disconnected	no charger voltage detected	< 1
connected	valid external voltage detected battery full	6..30 >= 12.03 (battery)
charging	valid external voltage detected battery needs charging	6..30 <= 11.4 (battery)
error	invalid external voltage detected ([1..6 Volt] or > 30 Volt)	1..6 or > 30

The charger component will continuously monitor the battery voltage. Charging starts if the battery voltage drops below the lower limit (12.5 Volt) and charging stops if the battery voltage reaches the higher limit (13 Volt). Once the battery has been charged and the charger stopped charging, the next charge cycle is only allowed after a pre-defined interval. This interval is defined in the board.h and is interval is not applicable when starting up or

when the charger voltage has been (re-) connected. By introducing the delay, we avoid charging the battery too often in a short period of time.

The charger has to be able to determine if a battery is not present or empty. In both scenario's the battery voltage will be 0V. To determine the presence of the battery, the charger will start charging (probe) and will verify if the battery voltage rises to 7.2V (fixed charger output voltage). If this is the case, the battery is not present. The charger will keep probing the battery every 10 seconds. In case the battery was present but empty, the battery voltage will be around 6V and the charger will start charging.

5.5.1 Status reporting

The charger state and charger voltage are communicated to the server every 15 min or when there is a state change or a difference in voltage bigger then 0.1V compared to the last message. As soon as the device is connected (connection status reported on the CAN-bus by a gateway e.g DNCM), and the time is synchronized it starts to publish messages. This way the device never publishes charger data immediate on powerup.

5.6 Configuration (`di_fw_config`)

5.6.1 Factory defaults (`di_fw_config_factory_defaults`)

After production of devices the low-level physical media is empty and factory defaults need to be loaded. During boot initialisation of the device get the value of the configuration item `DI_FW_CONFIG_FACTORY_DEFAULTS_ITEMS`. When it fails with the error `DNE_CHECKSUM`, it loads all configuration items default values. And writes the factory configuration default item flag to `false` (which results in an valid CRC). On next boot the read operation of the configuration item returns `DNOK` and the loading of defaults is skipped.

5.6.2 Self-test (`di_fw_config_selftest`)

For self-testing of the non-volatile configuration API and the low-level physical media a `DI_FW_CONFIG_SELF_TEST_ITEM` configuration item is created. The self-test will perform the item read, write and verify step. When one of the steps fails `!DNOK` errno value is returned. The write action uses a toggled value (`0x55`, `0xaa`) to make sure any physical media wear reducing algorithm is bypassed (read, check equal, write if not equal).

6. DI-Net CAN Bus Protocol (`di_can`)

6.1 Overview

The CAN Bus Protocol (DI-Net CAN) is a lightweight protocol designed to provide a highly reliable communication via the CAN bus.

The DI-Net CAN network is a decentralized peer network, where each peer (node) has a unique numeric identifier (Node ID).

The nodes of the DI-Net CAN network can communicate using any of the following communication methods:

- Publish - The primary method of data exchange with publish/subscribe semantics.
- Request/reply - The communication method for request/response, RPC-like interactions.

For each type of communication, a predefined set of data structures is used, where each data structure has a unique identifier - the Data Type ID.

Since every published message type has its own unique Data Type ID, and each node of the network has its own unique Node ID, a pair of Data Type ID and Node ID can be used to support redundant nodes with identical functionality inside the same network. Example: Given that both Node A and Node B publish some Message M, then other nodes that are interested in Message M can distinguish between the Message M from Node A and the Message M from Node B.

All nodes which implement DI-Net CAN need to set the CAN-bus baudrate to 1mbps.

6.2 Message publication

Message publication refers to the transmission of a serialized data structure over the CAN bus. Published messages are addressed to all nodes (except to the publisher itself) or a single node. This types of transmission is called broadcasting and unicasting.

Message publication is the primary way of data exchange for DI-Net CAN. Typical use cases may include transfer of the following kinds of data, either cyclically or ad hoc: sensor measurements, actuator commands etcetera.

A published message includes the following:

Field	Content
Payload	The serialized data structure
Data Type ID	The type of the data structure, i.e., how to interpret it
Source Node ID	The Node ID of the transmitting node
Destination Node ID	The Node ID of the receiving node
Transaction ID	A small integer ([0..15]) that is set to a fixed value boot time. The fixed value should be unique to allow the reassembler to reassemble frames that got interleaved when sent from a given

	node (refer to the CAN bus transport layer specification). For now the Transaction ID is derived from the (unique) node type

6.3 Core design goals

6.3.1 High throughput, low latency

How justified

The system of would typically run some high-frequency, (hard) real-time control loops, which necessitates the need for a low-latency, high-throughput communication channel.

How achieved

Low payload overhead, generally 0% to 12.5% per CAN frame. With a set speed of 1 mbps.

6.3.2 Democratic network

How justified

Since the network does not require a master node (bus controller), it has no single point of failure.

How achieved

Every node in the network has the same right to participate in the bus traffic.

DI-Net CAN does not require any network interaction for a node to begin functioning. For example, a sensor node can begin publishing measurements on the bus immediately after powering up.

6.3.3 Applications shall be able to exchange long datagrams transparently

How justified

Typical use cases for need to transfer coupled parameters, e.g., GPS solution, 3D vectors, etc., where each signal cannot be used independently. Such set of coupled parameters often does not fit into a single CAN frame (8-bytes), hence the need to split it into several CAN frames with a subsequent reassembly process on the receiving nodes.

Removing the decomposition/reassembly logic from the application renders it less error-prone.

How achieved

DI-Net CAN implements a simple transport layer using the 29-bit extended CAN ID field. Where every CAN frame has a follow-up number (frame-counter).

6.3.4 Common high-level functions should be clearly defined

How justified

The protocol should define and its implementations should implement such common tasks in order for the application designer to avoid reinventing the wheel for every application whenever the need for such common functionalities arises. Moving common functionalities into the library reduces the chances of the application designer from making a mistake.

How achieved

DI-Net CAN defines standard services and messages for the following:

- Network-wide time synchronization
- Network discovery

- Node configuration
- Network consistency checks
- Node status monitoring, which naturally grows into a device-wide health monitoring.

6.4 Transport layer

6.4.1 Transfer concept

As described in the chapter dedicated to the basic concepts of DI-Net Can, two methods of data exchange are used by the nodes of the DI-Net CAN network to communicate:

- Request-reply (unicast)
- Message publication (broadcast or unicast)

Both unicast and broadcast transfer distinguish between **single-frame transfer** (**SFT**) and **multi-frame transfer** (**MFT**). Single-frame transfer consists of a single CAN frame; it is used when there is no transfer payload. Multi-frame transfer consists of data decomposed into multiple CAN frames; it is used for exchanging long payloads.

When a node transmits a multi-frame transfer, all CAN frames of this transfer are pushed to the bus at once, in the proper order from the first frame to the last frame. The transport layer does not define any means of flow control for the following reasons:

- It would perform poorly with broadcast signal-based data exchange.
- Receiving nodes will not be able to process frames as they arrive anyway because all frames of a transfer must be received first before the encoded data structure can be de-serialized and checked for integrity. Thus, reception throughput is not limited by the application response time.
- Higher-level processes can implement flow control on top of the transport layer, if needed.

6.4.2 Transfer properties

A transfer comprises the following properties:

Header Fields	Content
Payload	The serialized data structure
Data Type ID	The identifier of the serialized data structure in the payload
Transfer Type	One of the following: • Reply • Request • Publish
Transaction ID	A small overflowing integer (described later in this part)

Multi-frame transfer also adds the following fields:

Field	Content
Transfer payload size	Size of the payload in bytes
Transfer payload CRC	Cyclic redundancy check of the transfer payload data

6.4.3 Transfer payload CRC

Transfer payload CRC is computed from the transfer payload in network byte order (MSB first). It uses CRC CCIT-16 with 0xffff as poly.

6.4.4 CAN frame format

As stated earlier, a transfer can be decomposed into one or more CAN frames. Only data frames of CAN 2.0B (29-bit ID field) are used.

6.4.5 CAN ID field

Every CAN frame uses the same fixed format of the extended CAN ID field (29 bits in total). The extended CAN ID field bit assignment, from high bit to low bit, is as follows:

Msg Type		**Data Type ID**										
28	27	26	25	24	23	22	21	20	19	18	17	16
00_2 - Raw		0	Raw Msg ID									
00_2 - Raw		1	Node Specific Raw Msg ID									
01_2 - DI-Net RPC		RPC Class ID [26:22]					RPC Method ID [21:16]					
10_2 - Net		Net Msg ID										
11_2 - Log		Log Msg ID										

Transfer type ID		**Frame Index**									**Transaction ID**				**Last frame**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00_2 – Reply error															
01_2 – Reply															
10_2 – Request															
11_2 – Publish															

Field	**Bit length**	**Note**
Msg Type [28:27]	2	Message type • 00_2 – Raw, Node Specific Raw • 01_2 – DI-Net RPC • 10_2 – Network monitoring and maintenance • 11_2 – Logging

Data Type ID [26:16]	11	Most significant bits; top priority during arbitration
		• For DI-Net RPC Msg Type the Data Type ID is split into:
		  ○ RPC Class ID [26:22]
		  ○ RPC Method ID [21:16]

Transfer Type ID [15:14]	2	• 00_2 – Reply error • 01_2 - Reply • 10_2 - Request • 11_2 – Publish
Frame Index [13:5]	9	Starting from 0 (start of message), until 2^9-1.

		Frames with the lowest index win in the arbitration, so they will be transmitted in order.
Transaction ID [4:1]	4	Least significant bits; lowest priority during arbitration
Last Frame [0]	1	• 0 - Expect more frames for this transfer • 1 - Current frame is the last frame of this transfer

6.4.6 Single frame transfer (SFT)

If there is no transfer payload (size is 0 bytes), the single-frame transfer is used. The entire CAN frame data field is dedicated for the source and destination Node IDs. Such allocation features zero payload overhead.

Data field contents of the first CAN frame

Source Node ID [0,3]		Destination Node ID [4,7]
CAN frame data field 8 bytes		
Byte 0	Bytes 1..6	Byte 7

6.4.7 Multi frame transfer (MFT)

If there is a transfer payload (size is 1 byte or more), the multi-frame transfer will be used. The first CAN frame is the same as the SFT. The second CAN frame data field will contain the Transfer Payload Size in the first two bytes in big-endian byte order (LSB first). The the next two bytes contain the Transfer Payload CRC in in big-endian byte order (LSB first); the rest will be used for the transfer payload. Maximum transfer payload data length

In order to unify the payload size limitation for all types of transfers, DI-Net CAN restricts the maximum possible transfer payload length to `(2^9) (maximum frame count) * 8 (can frame max bytes) – 4 (src nodeid) – 4 (dst nodeid)– 2 (crc) = 4086 bytes.`

Data field contents of the first CAN frame

Source Node ID [0,3]		Destination Node ID [4,7]
CAN frame data field 8 bytes		
Byte 0	Bytes 1..6	Byte 7

Data field contents of the second CAN frame (MFT metadata-only frame)

Transfer Payload Type (ptype) [07]	Transfer Payload Data Size (psize) [1,2]	Transfer Payload Data CRC (pcrc) [3,4]	Extended Metadata Flags (emflags) [5]	Extended Metadata (emdata) [6,7]
CAN frame data field 8 bytes				

Extended Metadata Flags field (emflags) (byte 5)

Bit	Name	Description
0	EM_DISABLE	The EM_DISABLE flag is used to mark the message doesn't use the Extended Metadata Flags and Extended Metadata fields. This is used for compatibility with older CAN-stacks which have those to fields set but unused to value 0xffffff. Otherwise all messages are processed incorrectly.
1	RT	The RT flag is used to mark the CAN message as Realtime. Realtime messages are unreliable proxied over UDP protocol. **NOTE:** When the RT flag is set the Extended Metadata field (emdata) is used to transport the uint16 Realtime message sequence number. It is encoded in network byte-order (big-endian).
2	Reserved	
3		
4		
5		
6		
7		
8		

When the Extended Metadata (emdata) field is unused it MUST be set to 0x0000.

Data field contents of the N-th CAN frame

Next 8 bytes of Transfer Payload Data		
CAN frame data field 8 bytes		
Byte 0	Bytes 1..6	Byte 7

Data field contents of the Last CAN frame

Last N-bytes of Transfer Payload Data		
CAN frame data field N bytes		
Byte 0		Byte N

6.4.8 Transfertype request/reply

Request/reply is a two-step data exchange between two nodes.

Request/reply sequence:

1. The client sends a Request transfer to the peer (Source Node ID is the client's Node ID; Destination Node ID).
2. The peer application takes appropriate actions and returns the response data.

The client will need to match the response with the corresponding request using the following set of transfer fields:

- Data Type ID

- Destination Node ID (local Node ID)
- Source Node ID (Node ID of the remote endpoint).
- Transaction ID

It is easy to see that the peer node is required to send the Reply transfer with the same value of the Transaction ID as it was received.

For Request transfers, the Transaction ID value shall be incremented by the transmitting node with every outgoing transfer of the given Data Type ID and Destination Node ID.

6.4.9 Transfertype reply error

When a node sends a request to another node an error can occur. Because the transfer payload data only contains valid data a separate transfer type is created to reply a error.

6.4.10 Message publication

Message publication, either broadcast or unicast, is a one-way data exchange from one node to all other nodes in the network (broadcasting) or to exactly one remote node (unicasting). Receiving nodes cannot provide any kind of feedback nor can they perform any flow control.

For message publication transfers, the Transfer ID value must be incremented by the transmitting node with every outgoing transfer.

6.4.11 Transfer reception

A DI-Net CAN bus can accommodate no more than one unique transfer at a time. A transfer is considered unique if there are no other transfers with the same values for the following properties:

- Msg Type ID
- Data Type ID
- Transaction ID

6.4.12 Node ID and Data Type ID

Node ID `0x0 (unknown)` and `0xffffffff (broadcast)` are reserved. It is used by protocol implementations to represent the broadcast Node ID and to check if nodes already have strapped a unique node id. The resulting usable range is 1.. 2^{31}-2. The range of Data Type ID is 0..2047 (2^{11} - 1).

The Node ID must be derived from a 128-bit hardware unique ID hex-encoded string (32-byte without null termination) (combination of chip unique id, vendor id, etc). The 128-bit hardware unique ID is then hashed into a 32-bit number using MurmurHash2.

6.4.13 CAN bus requirements

DI-Net CAN is bit-rate agnostic, so, technically, any bit rate can be used as long as it is supported by the physical layer. However, the application designer should follow the hardware design recommendations.

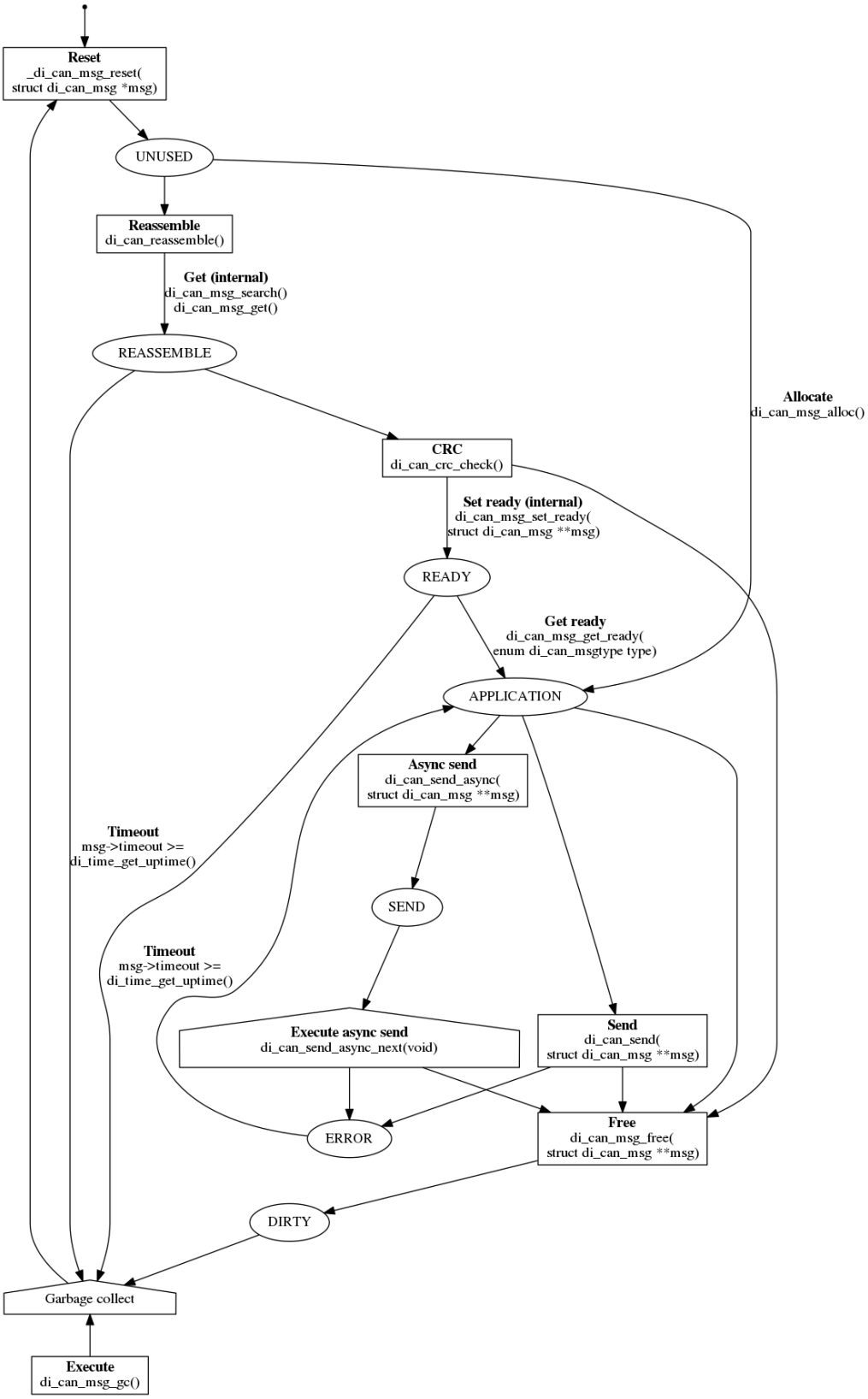
The CAN bus driver and CAN controller should consider using priority inversion free transmission, wherein the CAN frame transmission order is defined by the CAN arbitration rules rather than by the order of appearance in the TX buffer.

6.5 Message dictionary

The available messages (Raw/RPC/Net/Log) are described in the CAN Messages Dictionary document. See ref[3].

6.6 Protocol stack message states (`di_can_msg_state`)

The CAN messages are state-full in the CAN-stack. As shown in the figure below:



6.6.1 DI_CAN_MSG_STATE_UNUSED

The message is unused and can be request by the CAN stack or by the application.

6.6.2 DI_CAN_MSG_STATE_REASSEMBLY

The message is in use by the reassembler. It can be garbage collected when the message is not completed in time.

6.6.3 DI_CAN_MSG_STATE_SEND

Message is in queue for next asynchronous send operation.

6.6.4 DI_CAN_MSG_STATE_READY

The message is reassembled and the CRC is valid and can be read by the application. When the message is not read in time it can be garbage collected by the can stack.

6.6.5 DI_CAN_MSG_STATE_APPLICATION

The message is in use by the application. And is NEVER garbage collected by the can stack.

6.6.6 DI_CAN_MSG_STATE_DIRTY

The message is marked dirty so on the next run of the garbage collector it is reset and set UNUSED.

6.6.7 DI_CAN_MSG_STATE_ERROR

The message is in an error state. This is used in asynchronous message sending or a reply timeout has occurred.

6.7 RAW messages (DI_CAN_MSGTYPE_RAW)

The raw message type is used for node-to-node interaction (e.g requesting device unique id string). The details are depicted in the CAN Messages Dictionary document, see ref[3].

6.8 RPC messages (DI_CAN_MSGTYPE_RPC)

RPC messages are the messages received/send from/to the backend. These must always have the MessagePack datatype (DI_CAN_DTYPE_MSGPACK). The details are depicted in the CAN Messages Dictionary document, see ref[3].

6.8.1 Publish & Publish Realtime (RT)

Destination node ID must always be set to broadcast.

Publish messages on the CAN bus may contain the DI-Net RPC result field array (See ref[2] – 4.1 Request object) as payload. The "[class]:[methodname]" is already in the CAN frame header as Data Type ID.

Messages can be sent Realtime and distributed over multiple channels (e.g DNCMs). A sequence number is necessary for (de)duplication when sent or received over multiple channels. The Realtime message has the RT bit set in the Extended Metadata Flags (emflags) field. The Realtime message sequence number (uint16) is placed in the Extended Metadata (emdata) field. When implemented by an DI-Net RPC gateway (e.g DNCM) the flag and sequence number is translated between DI-Net RPC message header and DI-CAN MFT metadata frame (see subsection 6.4.7).

6.8.2 Request/reply

Request-reply messages are mostly relayed from a communication module (e.g. DNCM) to and from the CAN bus. A request is always broadcasted.

Request messages on the CAN bus may contain the DI-Net RPC param field object (See ref[2] – 4.1 Request object) as payload. The "[class]:[methodname]" is already in the CAN frame header as Data Type ID.

A RPC reply only contains the DI-Net RPC result field (See ref[2] – 4.2.1 Error reply).

6.8.3 Request/reply error

When a request is broadcasted and the node is unable to satisfy the request a reply error (Transfertype) is send. The RPC error reply may contain the DI-Net RPC error field object (See ref[2] – 4.2.1 Error reply) as payload.

6.9 Network maintenance and monitoring (DI_CAN_MSGTYPE_NET)

Features

- Node discovery and monitoring
- Node redundancy

6.9.1 Definitions and enumerations

This section describes the definitions and enumerations for the network maintenance message type.

6.9.1.1 Node types (`enum di_can_net_node_types`)

Nodes on the network need to distinguish each other, this is based on type.

Enumerator (DI_CAN_NET_NODE_TYPE_*)	Value (uint8_t)
UNKNOWN	0x00
GATEWAY (e.g DNCM which relays RPC messages to/from server)	0x01
DEVICE	0x02

6.9.2 Network discovery (`di_can_net_discover`)

The network is discovered by sending a broadcast request message (DI_CAN_NET_DTYPE_DISCOVER). Other nodes then reply with their current node type (see 6.9.1.1) and node role (see 6.9.4). This messages layout is depicted in ref[3].

6.9.3 Node states and events

To monitor and act on node changes every node has its state tracked and node events are emitted accordingly.

Enumerator (DI_CAN_NET_NODE_STATE_*)
UNKNOWN
ONLINE, Node is online (last seen within a given time)
OFFLINE, Node is offline (not seen within a given time)

Only leader nodes (within same node type) are active monitored and will emit the leader offline event. Non-leader nodes will not transition into the offline state as they are not active monitored with the DI_CAN_NET_DTYPE_HEARTBEAT message.

The application is able to subscribe to all events, on all nodes using a call back function. In the table below the possible events are depicted:

Enumerator (DI_CAN_NET_NODE_EVENT_*)
NEW, New node detected
ONLINE, Node becomes online
LEADER, Node becomes leader
LEADER_OFFLINE, Leader-node goes offline (due to heartbeat timeout)
LEADER_TAKEOVER, Self node takes over other leader
CONFLICT, Node is in conflict (duplicate nodeid, multiple leaders of same type)

6.9.4 Passive, follower and leader node roles (`enum di_can_net_node_roles`)

All nodes in the DI-Net CAN network fulfil a role. This necessary to detect high-available peer-nodes (within the same type). The idea and naming is taken from the Raft Consensus Algorithm (see ref).

The known roles are depicted in the table below:

Enumerator (DI_CAN_NET_NODE_ROLE_*)	Value (uint8_t)
UNKNOWN	0x00

`PASSIVE`, Node is standalone (or not fully initialized yet)	0x01
`FOLLOWER`, Node follows a leader node with same type	0x02
`LEADER`, Node is leader within same types	0x03

When a node boots and the DI-Net CAN stack is initialized the net subsystem defaults self to the passive role. Passive nodes only respond to network discovery requests, and will not fulfil a role in a redundant configuration.

A node when completely boot it can only set its own role to follower. It is not allowed to set it to leader from the application. As the leader role is automatically assigned. The application may "downgrade" its own role from leader to follower when necessary.

In a high-available/redundant configuration nodes are able to take over other nodes (of the same type) when their state transitions to offline.

Leaders publish a heartbeat (`DI_CAN_NET_DTYPE_HEARTBEAT`) with a given interval, follower (of the same node type) will only monitor the heartbeat of leaders.

6.9.5 Periodic execution of NET subsystem (`di_can_net_execute`)

The DI-Net CAN NET subsystem needs to be periodically executed to maintain all internal states and decide if nodes went offline (due to not seen timeout). The network is always discovered. The multi-depth FSM is depicted in the figure below:

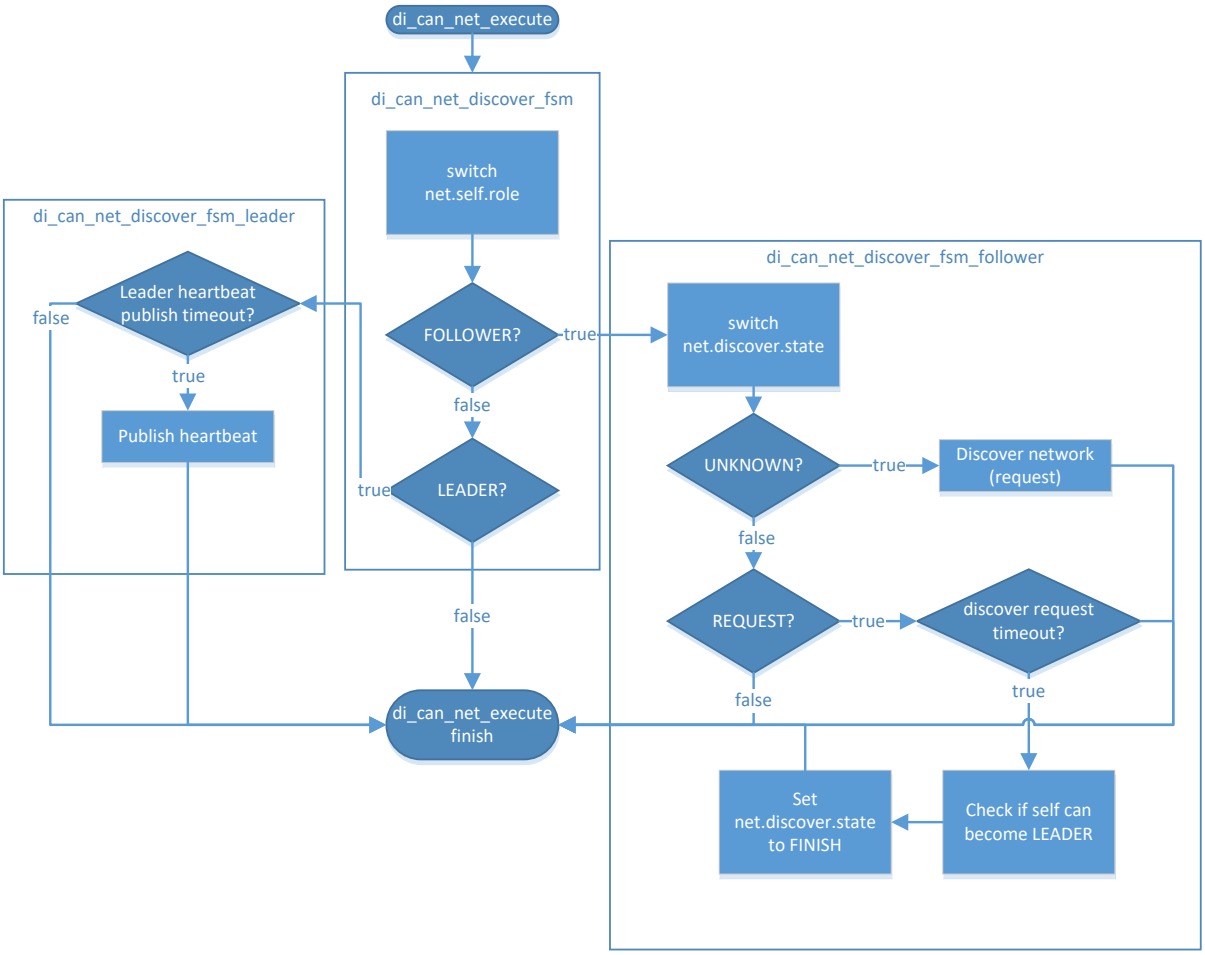


Figure 6 DI-Net CAN net subsystem periodic execute (di_can_net_execute)

Node replies and heartbeats (message receive) is handled by a separate thread which runs the correct callbacks.

6.10 Logging (`DI_CAN_MSGTYPE_LOG`)

The logging message type is used to log events and errors. The message payload data is MessagePack encoded. It contains key-value pairs with DI-Net timestamp, loglevel, component (library/application) and module identifier. The log message is always a human readable string. The details are depicted in the CAN Messages Dictionary document, see ref[3].

For more details about the logging facility (`di_log`) see Section 4.6.

7. DI-Net Communication Module (DNCM)

This chapter is an outline of the architecture and design of the DNCM firmware.

7.1 Hardware Architecture

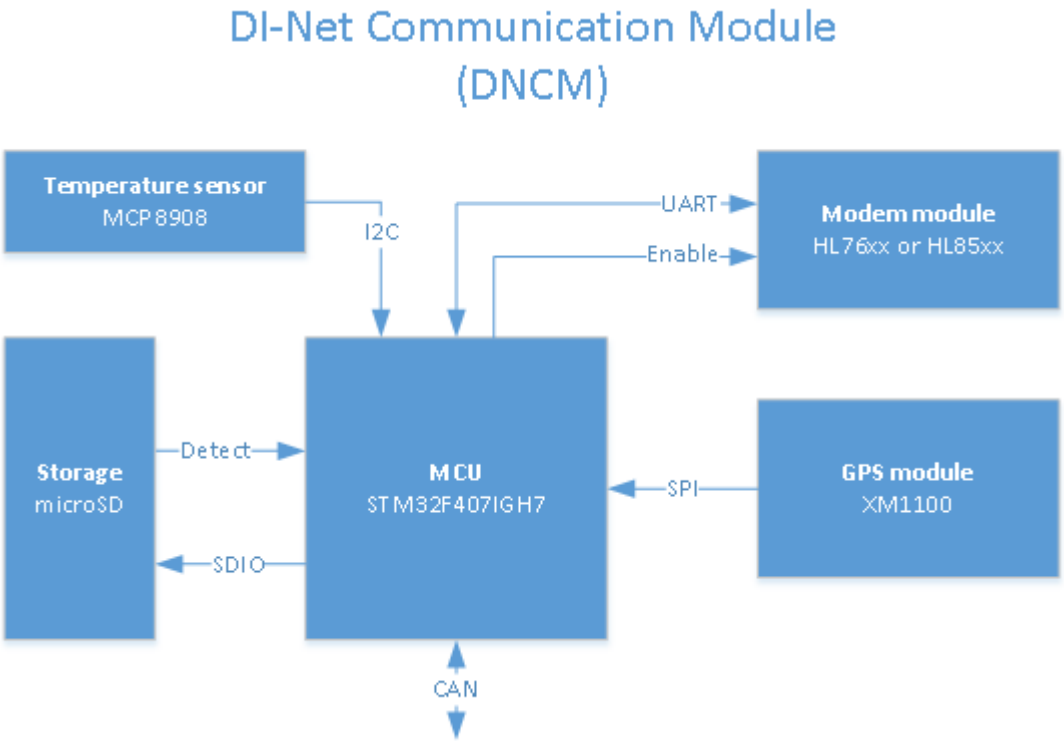


Figure 7 DNCM hardware architecture

The MCU is responsible for the following tasks:

- Communicate and relay information to a centralized server using the modem module
- Communicate with other CAN-bus peripherals (e.g. DUU mainboard, other DNCM etc.)
- Report DNCM temperature with the onboard temperature sensor
- Measure and report GPS location as registered CAN-bus device unique ID

7.2 Modem driver (`di_drv_hl854x`)

The driver is implement according to the AirPrime HL6 and HL8 AT command set specification see ref[5] and HL76xx see ref[7]. The driver implements functionality for the HL8548 (3G) and HL769x (4G) model. Because the DNCM firmware uses the ChibiOS RTOS the driver is used from multiple threads and is drawn the architecture is shown in the figure below:

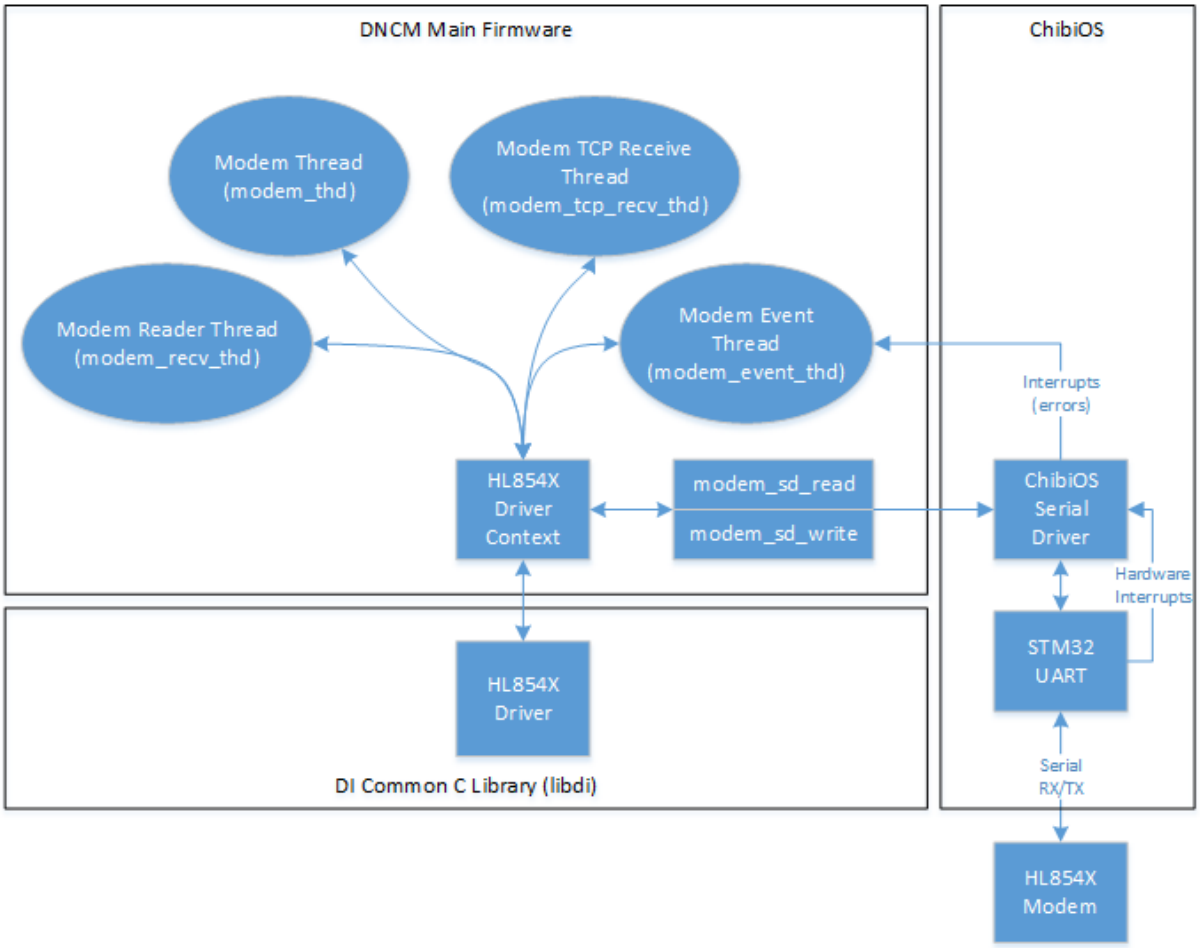


Figure 8 DNCM Modem Tasks

7.2.1 Modem FSM thread (`dncm_modem_fsm_thd`)

The modem FSM thread task initializes and enables the peripheral in the modem accordingly using the respecting state machines:

- SIM state machine, SIM card selection and pin authorisation
- Net state machine, network registration
- Con state machine, data connection setup
- GPS state machine, global positioning and time
- TCP state machine, TCP connection
- UDP state machine, UDP listening server configuration

7.2.2 Modem reader thread (`dncm_modem_reader_thd`)

The modem reader thread will read data asynchronous from the application. Incoming data is read when the `CHN_INPUT_AVAILABLE` event is triggered from the ChibiOS Serial Driver. The data chunks are fed into the HL854X driver. The thread also registers for ChibiOS Serial Driver/UART events when errors occur.

7.2.3 Modem TCP receive thread (`dncm_tcp_recv_thd`)

The Modem TCP receive thread will wait for available data on the TCP connection. And when there is data available it is read into an available receive buffer.

All data which comes from the TCP connection is always a request, a reply on the DNCM is generated or passed to the CAN-bus (see section 7.3).

7.3 Request-reply bridging of TCP and CAN-bus

The DNCM bridges request from the server at the other site of the TCP connection to the CAN-bus. It keeps track of outstanding requests and will timeout if the reply is not within a time window of 5 seconds satisfied from the CAN-bus. There is a mapping between the server RPC message IDs (`uint32_t`) and the actual DI-Net CAN message send (`canid`). In the figure on the next page the architecture of the message bridging/flow in the whole context is drawn.

The request-reply bridge will only accept one outstanding RPC request with "`req`" : "`<class>:<method>`". When a reply is still outstanding and a request is performed on the outstanding "`<class>:<method>`" the DNCM firmware will respond with an error code: `` `DNE_BUSY``, code 5, "Device or resource busy"`. This design decision makes the implementation very simple and robust.

When a request is successfully bridged the device will respond with a result array or an error object (See ref[2]).

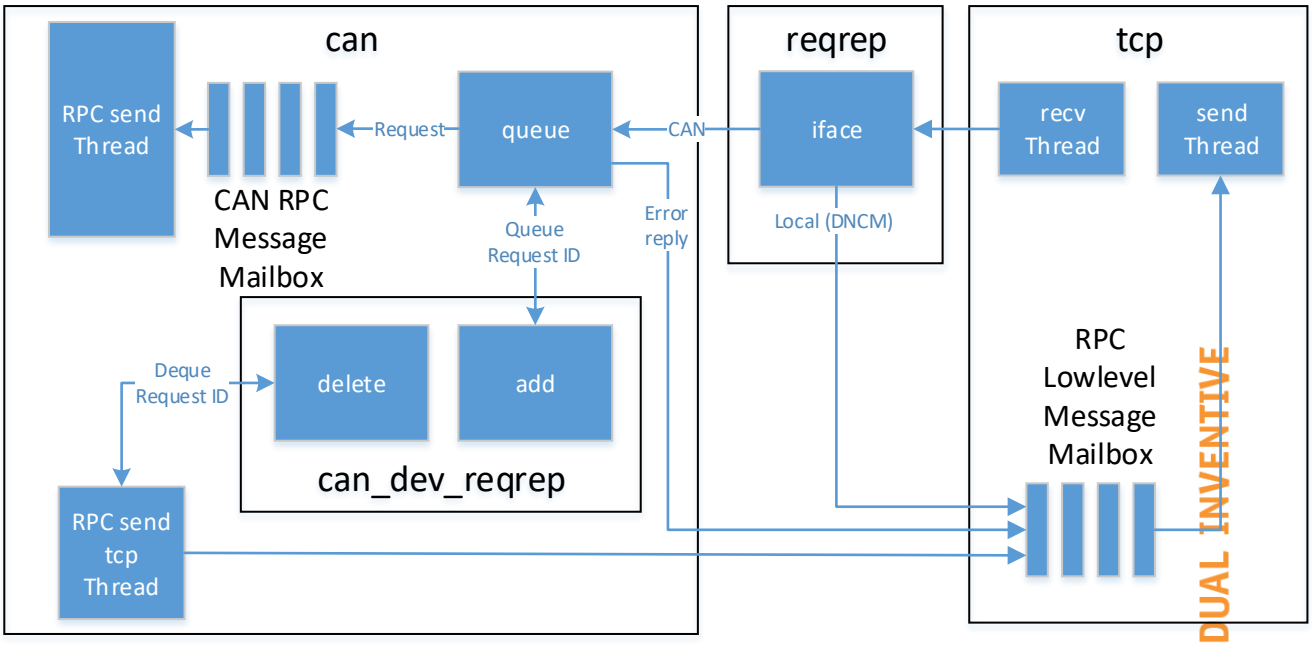


Figure 9 DNCM RPC request message routing

7.3.1 Transaction from TCP to CAN-bus

1. Make sure there is a free slot in the outstanding request queue
2. Check if DI-Net RPC request ID is not already in the queue (collision)
3. Calculate CAN-bus ID from DI-Net RPC request ID
4. Save both IDs and the deadline of timeout
5. Convert RPC message to CAN RPC message
6. Put message in the mailbox for the CAN RPC send thread

7.3.2 Transaction from CAN-bus to TCP or UDP (RT message)

1. Check if DI-CAN RPC message is a reply
   - Check if there is an outstanding reply which matches a DI-Net RPC request ID
2. Check if DI-CAN RT flag is set
   - Load the RT sequence number from DI-CAN RPC message into DI-Net RPC message
3. Convert DI-CAN RPC message to DI-Net RPC low-level message
4. If DI-CAN RT flag is set
   - Send DI-Net RPC message direct over UDP
   - Release the TCP buffer
5. If DI-CAN RT flag is unset
   - Put message in the mailbox for the TCP send thread
   - Remove outstanding entry from the request queue

7.4 Pub-sub messaging between UDP and CAN-bus (`dncm_udp`)

The DNCM creates a listing UDP socket pre-defined on port 4000. Realtime DI-Net RPC low-level messages are sent and received from this socket. The DI-CAN protocol stack is Realtime-aware and uses the DI-Net RPC real-time message sequence number ("`rt:seqnr`" property from the header) and is exchanged between the CAN-bus and UDP protocol. See also DI-CAN subsection 6.4.7.

7.5 Watchdog monitor (`dncm_monitor`)

A watchdog monitor (thread) is implemented in case of a software or external hardware failure will result in an automatic reboot of the device.

The HL854x modem is a complex device and unhandled errors can possible result in firmware misbehaviour. To mitigate the risk of connection hangs the TCP connection is monitored when it first comes online.

In the list below the following mechanisms are monitored:

- Check if the modem reacts on AT-commands 10 seconds after boot and auto-baud and SIM card is ready.
- Check if the modem has a TCP connection 60 seconds after boot
- Check if the modem TCP-connection has data flowing every 10 seconds when the TCP connection is UP and the proxy has replied the RPC handshake

7.6 SDCard blackbox logging (`dncm_sdcard`)

The SDCard is used to log CAN-bus `DI_CAN_MSGTYPE_LOG` messages.

The blackbox logging writes the following files:

- `di_log_00000000000000.bin`
  - When time is not available yet `(>= year 2016)`
  - File is renamed after an absolute time has been synced (TCP/GPS)
- `di_log_<year><month><day><hour><minute><second>.bin`
  - Every log file starts with a magic string (with ending 0 character):
    - "`DINet-LogFile.v1`"
  - Every log message starts with a magic string (with ending 0 character):
    - "`LMsg.v1`"

7.7 Redundancy (DI-Net CAN node redundancy)

The DNCM firmware uses the high-available follower-leader node roles from the DI-Net CAN protocol (see section 6.9.4).

7.7.1 Firmware boot and passive role

At firmware boot the DNCM uses the passive role (see section 6.9.1.1). It will set its type to gateway (see section 6.9.4). In the passive role there is no TCP connection, hence there are no RPC messages relayed between TCP and the CAN-bus.

7.7.2 Follower role

When the TCP connection to the server is established and the DI-Net RPC lowlevel handshake reply is received the role is changed from passive to follower. In the follower role no RPC messages are relayed between TCP and the CAN-bus.

When a request is received from the server (in the follower role) it is always replied with error code `DNE_BUSY`. Also for "`dncm:<method>`" RPC messages.

7.7.3 Leader role

The leader role is automatically assigned by the DI-Net CAN-stack when no other leaders are active or a leader goes offline. In the leader role the DNCM relays RPC messages between TCP and the CAN-bus.

7.7.4 Leader role conflict

When a leader detects a conflict with another leader on the CAN-bus it raises a DNE_FIRMWARE_LEADER_CONFLICT device error and reports it to the server in the `device:data` heartbeat.

Currently there is no recovery from the DNE_FIRMWARE_LEADER_CONFLICT error. When this error is raised the behaviour of message routing by the DNCMs is undefined. Both DNCMs then could have a connection as the CAN-bus `device:uid`. It is possible RPC publish messages are send from both DNCMs and replies may not be delivered.

7.8 DI-Net RPC `device:uid` handshake and register

The DNCM registers itself to the MTinfo 3000 backend using the DI-Net RPC Lowlevel Protocol. Its own `device:uid` is derived from the STM32 96-bit unique id and chipid. After a TCP connection is established a handshake request (`DNP_HS_REQUEST`) is send.

For the device on the CAN-bus a separate `device:uid` is used. This `device:uid` is requested with the a DI-Net CAN Raw `DEVICE_UID` message. And is cached into the RAM of the DNCM. Currently the CAN-bus is limited to only have one registered `device:uid`. The CAN-bus `device:uid` is registered using the `DNP_REGISTER` DI-Net RPC Lowlevel message.

7.9 CAN device module (`dncm_can_dev_*`)

The DNCM CAN device functionality is split into multiple modules which are necessary for the DNCM to establish an RPC connection (over TCP) as the CAN device:uid. The following modules are created:

- uid: CAN device:uid cache and RPC registration
- reqrep: CAN <-> TCP request reply handler
- gps_sensor: GPS sensor reporting
- comm_status: Communication status reporting

7.9.1 dncm_can_dev_uid Module

The uid module is used for:

- Requesting and storing the CAN device:uid;

- Storing the DI-Net RPC device:uid registration state;

7.9.2 dncm_can_dev_reqrep Module

The reqrep module consists of:

- A request setter which stores the request id and the rpc message type
- A request getter which gets the request id based on the rpc message type
- A periodic task to send timeout replies (of the CAN device doesn't send a reply in time)

7.9.3 dncm_can_dev_gps_sensor Module

The gps_sensor module consists of:

- A configuration function for controlling the GPS sleep and reporting interval (from the CAN-bus)
- A function to trigger a GPS update and report the result (from the CAN-bus)
- A periodic task which checks for changes and sends the updates

7.9.4 dncm_can_dev_comm_status Module

The comm_status module consists of

- A configuration function for controlling the Communication Status reporting on-change and interval (from the CAN-bus)

7.10 Time synchronisation

Absolute time is synchronized as soon as the DNCM has a TCP connection to the server. A DI-Net RPC lowlevel time sync request is sent and the time is used as the new absolute time + latency. The roundtrip latency is calculated as follows: (recv_time – send_time/2).

GPS time is not used as a time source.

7.11 Temperature sensor driver (`dncm_mcp8908`)

The MCP8909 temperature sensor measures the ambient temperature on the DNCM. And works as follows:

- Initialisation
  - The sensor is probed over I2C and checked for the correct device and manufacturer ID
  - Configuration register is written to its default value (0x0000)
  - Resolution register is written to use 0.125 degree resolution with a typical conversion time of 130ms
- Temperature read
  - Read the Ta register 16-bit value
  - Convert the register value to degrees Celsius

7.12 Temperature reporting (`dncm_temp`)

The DNCM reports its onboard temperature with a periodic interval of 15 minutes as its own device:uid. It uses the MCP8909 sensor (see section 7.11). The periodic task performs the following actions:

  - Measure MCP8909 temperature

- Allocate a RPC message buffer and write the message as
  - Classmethod: `device:data`
  - UID: 100 (label: `dncm-temp`)
- Send the message over TCP to the server as DNCM device:uid

7.13 GPS

The DNCM GPS functionality is divided into multiple modules. It consists of a low-level driver for the XM1110 chip (`dncm_xm1110`). The middleware (`dncm_gps`), and the registered CAN `device:uid` GPS sensor (`dncm_can_dev_gps_sensor`). In the figure below the architecture is depicted:

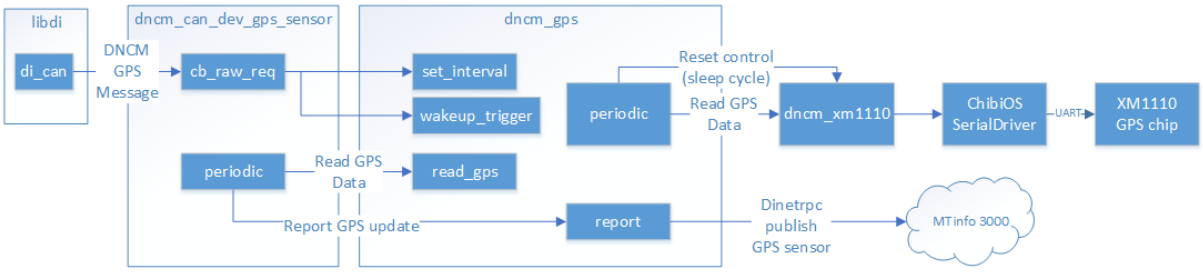


Figure 10 DNCM GPS architecture

7.13.1 XM1110 driver (`dncm_xm1110`)

The XM1110 driver reads data from the XM1110 in its own thread over the USART Serial Driver and decodes the GNGGA frames. The driver also allows middleware to control the hardware reset of the chip.

7.13.2 DNCM GPS (`dncm_gps`)

The DNCM GPS module reads from the `dncm_xm1110` driver to retrieve the current GPS location. The `dncm_gps` module controls the sleep cycle of the XM1110 with the hardware reset to save power. At DNCM startup the XM1110 is turned on and polled with a `di_fw_periodic task` with an interval of 1 second for a GPS update. It is put into sleep when an update is read. The next wake time is calculated based on the configuration of `dncm_can_dev_gps_sensor` module. When no interval is set or an update is triggered from it will sleep the XM1110 until the application decides.

The application may trigger a GPS update before the next wake time is reached. After a GPS update is read the next wake time is calculated and the previous next wake time is overwritten.

7.13.3 CAN device:uid GPS sensor (`dncm_can_dev_gps_sensor`)

A node on the CAN-bus must register the `device:uid` to the DNCM on which the GPS sensor is reported. This `device:uid` is used for reporting the GPS sensor to MTinfo 3000.

The DNCM listens on the CAN-bus for GPS sensor control messages (wakeup trigger, sleepcycle interval configuration).
The message handler `dncm_can_dev_gps_sensor_cb_raw_req` is registered for messages with datatype ID `DNCM_GPS_SENSOR_TRIGGER` (see ref[3]).

A periodic task with an interval of a second to poll-reads the `dncm_gps` module for GPS updates. When an update is ready it is reported to MTinfo 3000 using `dncm_gps_report`.