LiquidCan Protocol Documentation

for Distributed Embedded Systems
at the TU Wien Space Team

TU Wien Space Team

Version ~~1.3~~1.4

March 5, 2026

Version History

Version	Date	Changes	Author
0.1	2025-12-16	Initial protocol specification proposal	Raffael Rott
0.2	2025-12-16	Implemented Proposals from Team members	Raffael Rott
0.3	2026-01-05	Fixed final bugs before initial release	Raffael Rott
1.0	2026-01-11	Fixed minor typos.	Raffael Rott
1.1	2026-01-25	Fixed total sizes of structs.	Fabian Weichselbaum
1.2	2026-02-09	Fixed inconsistant field value lengths & typos	Raffael Rott
1.3	2026-02-28	Added a status field to field access responses	Raffael Rott
1.4	2026-03-05	Variable-length frame serialization; updated payload sizes	Michael Debertol

Contents

1 Purpose and Scope

The purpose of the LiquidCan protocol is to serve at the heart of all future liquids projects at the TU Wien Space Team. Building on the CAN FD standard, it defines how our client devices (such as ECUs) communicate with each other and the central server and with each other. It is designed to be as simple and extensible as possible. One of the design goals of this protocol is to minimize the amount of common type or field definitions between the server and the nodes.

2 Notation and Conventions

- This protocol uses the CAN-FD extension
- All fields are little endian.
- All strings are ASCII encoded and null terminated.
- Payload length: Variable depending on message type but up to 64 Bytes.

2.1 Common Terms

Term	Description
(CAN) Client	A CAN client is a device which is connected to the bus
Node	Every client which is not the server
TelemetryValue	A telemetryValue is a non-externally modifiable value which periodically gets sent to the server
Parameters	Parameters can be externally modified
Field	An encompassing term for telemetryValues and parameters
ECU	A commonly used embedded CAN device at the TU Wien Space Team

3 CAN Identifier Scheme & NodeID

Each device on the bus has its own unique Node ID. The Server is assigned the Node ID 0. The CAN ID is composed of 11 bits. It contains the sender and receiver Node IDs and a priority bit.

Note the location of the priority bit: It is set as the last bit here since this document expects little-endianness. On the actual bus the priority bit will be sent first, therefore ensuring that the packets are properly prioritised by the CAN Protocol.

Field	Bits	Description
Receiver	5	Destination node ID
Sender	5	Source node ID
Priority	1	Message priority (0=low, 1=high)
Total	**11**	Standard CAN ID

4 Common Frame Layout

The CAN data field ~~consists~~ is variable-length, up to a maximum of 64 bytes. The first byte of each message contains the message type. The remaining bytes carry the payload, whose length depends on the message type. This simple format allows the protocol to be extended in the future by adding more message types. See Section 9 for a detailed description of message types and their numeric values.

Field	Bytes	Description
message_type	1	Type of message (see Message Types)
data	0–63	Payload data (length depends on message type)

Note: Each message type defines its own payload size. Message types that contain only fixed-size fields have a constant wire size, while those that end with a variable-length field (e.g. null-terminated strings or packed data arrays) may use fewer bytes than the maximum. Should the payload not fit exactly into one of the CAN FD data field sizes (1-8, 12, 16, 20, 24, 32, 48, or 64 bytes), it should be padded with zeros to the next valid size.

5 Node Registration

As soon as a node comes online (or when it receives a `node_info_req`), it enters the registration phase. See Figure 1 for a graphical representation the whole registration procedure. Here it sends out a `node_info_announcement`. This announces the node to the bus and includes its name, number of telemetryValues and parameters, and its firmware version through the firmware and LiquidCan hashes (see 10.1). The central server registers the new node and waits for `field_registration` messages. From this point on the node is able to send/receive messages on the bus.

6 Field Registration & Management

Fields are the heart of the protocol. The term Field serves as a general term for both telemetryValues and parameters. TelemetryValues are periodically sent to the server and non-modifiable. They are meant to represent sensor data or other information which should be periodically logged. Parameters can be externally modified and locked. These are meant for configuration fields, modifiable by either the server or other nodes.

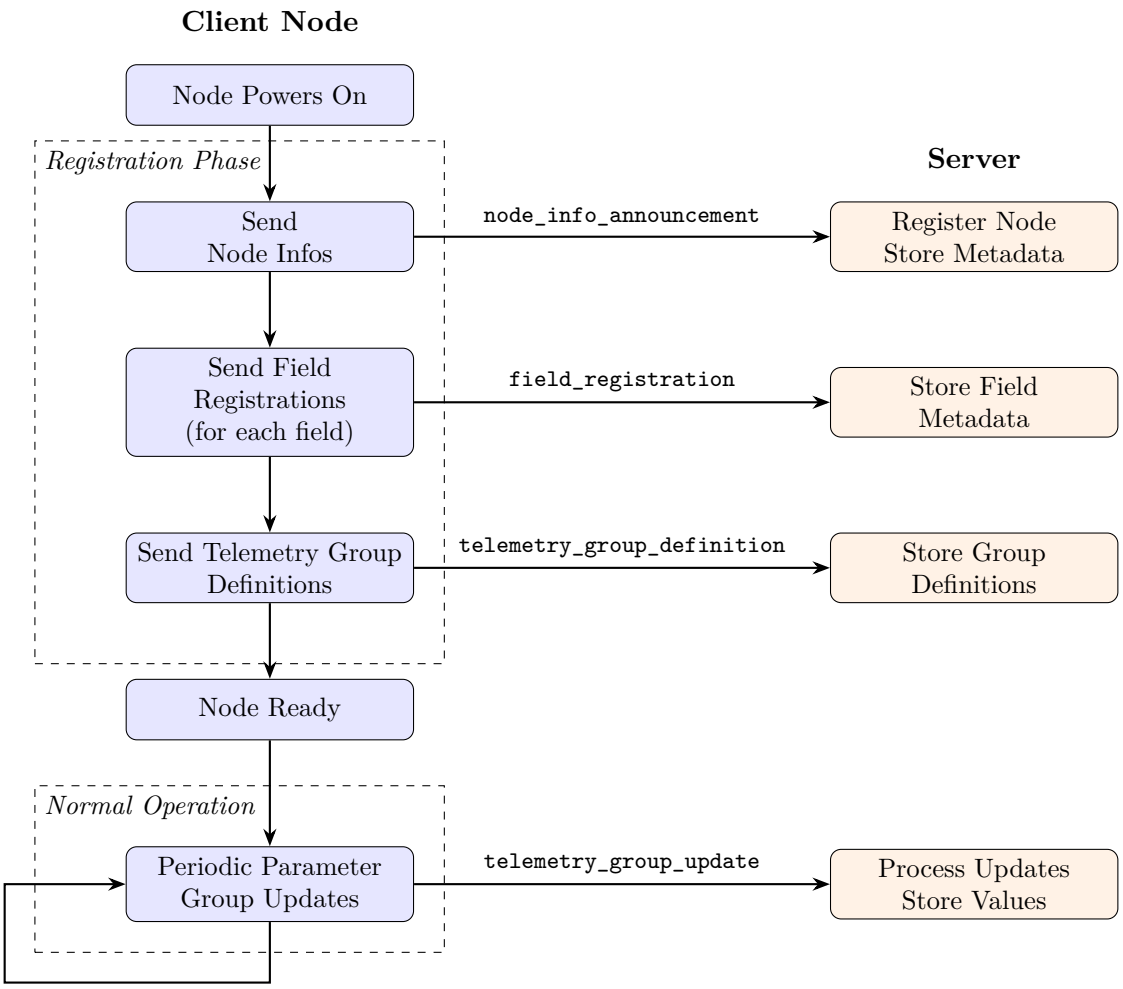

Figure 1: Node Registration and Field Registration Flow

6.1 Registration

TelemetryValues and parameters are dynamically defined over the bus.

After registration was initialized, a node sends out `field_registration` messages, one for each parameter/telemetryValue. The FieldRegistration includes a field ID, the datatype of the field, and a human-readable name. The name and field ID have to be unique per node. The first bit of paramter field IDs is 0 and the first bit of telemetryValue field IDs is 1. From this point on, the server knows which fields the node has. The fieldID `0` is reserved to indicate an invalid fieldID. It must not be assigned to a field. FieldIDs must be assigned in a determenistic way. This ensures that named fields have the same fieldID between reboots.

Next, the node sends `telemetry_group_definition` messages. The node sends a group ID to identify the group and a list of field IDs. The order of the field IDs must be the same as the order of fields in future `telemetry_group_updates`. The node must ensure that the values of all of the defined telemetryValues can fit into a TelemetryGroupUpdate.

6.2 Regular Operation

6.2.1 Parameter Updates

During regular operation, each node sends `telemetry_group_update` messages at a defined interval. The order of values here must match the field id ordering of the corresponding `telemetry_group_definition`. The interval can vary between groups, allowing nodes to send telemetryValues at different intervals to, for example, reduce bus utilization.

6.2.2 Parameter Setting

Other bus members can send a `parameter_set_req` message, which includes the field ID of the parameter and the new value. Once the node receives the request, it attempts to set the internal parameter value and responds with a `parameter_set_confirmation` message containing a status code, a field status, the parameter ID, and the actual value. The status field indicates whether the operation was successful or provides an error code (see ParameterSetStatus enum in 10.9). The `field_status` indicates whether the parameter exists. If the parameter was not found (field_status = NotFound) the status field should return InvalidParameterID. Otherwise, the value field should contain the actual value read back from the parameter and not simply the value that was received in the request.

When a parameter is internally modified through some automated system, the updated value must be sent as a `parameter_set_confirmation` message to the server with a `NodeToNodeModification` status code.

6.2.3 Parameter Locking

A parameter can optionally be locked through a `parameter_set_lock_req` message . After a parameter has been locked, it cannot be modified by an external node. A parameter can only be unlocked by the locking node or the server.To lock a parameter, a node sends a `parameter_set_lock_req` with the fieldID and the locking status (0=unlocked, 1=locked). The recieving node responds with a `parameter_set_lock_confirmation`, confirming the sent fieldID, the locking status, and a field status (see FieldStatus enum in 10.8). If the parameter is not found the node should respond with a not found status(field_status = NotFound) and the erroneous field id. The receiving node also sends a `parameter_set_lock_confirmation` to the server to update it on the locking status of the parameter. When a parameter was not found, it does not communicate this with the server.

6.2.4 Requesting Field Data

A field can be accessed through a `field_get_req` message, which contains the field ID. Nodes respond with a `field_get_res` message, containing the field ID, a field status (see FieldStatus enum in 10.8), and the value of the field. If the field is not found (field_status = NotFound), the field_id should still contain the requested field ID from the request message, and the value field content is undefined.

6.2.5 Field name Lookup

The field name lookup covers the case where nodes need to access fields from other nodes. Since they don't recieve the `field_registration` messages, they don't know the fieldIDs of the named fields they want to access. `field_id_lookup_req` messages contain the remote field name. The Node responds with a `field_id_lookup_res` message, containing the fieldID, the datatype of the field, and a field status (see FieldStatus enum in 10.8). If the field is not found (field_status = NotFound), the field_id should still contain the requested field ID from the request message.

7 Heartbeats

Heartbeats ensure that the system does not reach a state where it is still dangerous to physically handle but not accessible through CAN messages. The `heartbeat_req` message sent from the server contains a continuously increasing counter. The counter value is unique to each node. If a node does not receive `heartbeat_req` messages, it will default to a safe state. Similarly, the server takes note of any unresponsive nodes.

8 Status Messages

Nodes may send optional status messages to the server. Currently, the three status message types are: `info_status`, `warning_status`, and `error_status`, as defined in section 9. Each message contains a null-terminated string with a status message.

9 Message Types

The following message types are defined in the protocol:

Enum	Message Type	Data Payload	Direction	Description
Node Discovery and Information				
0	`node_info_req`	No payload	Server → Node	Request node informati
1	`node_info_announcement`	NodeInfoRes (10.1)	Node → Server	Response with node cap identification
Status Messages				
10	`info_status`	Status (10.2)	Node → Server	Informational status m
11	`warning_status`	Status (10.2)	Node → Server	Warning status messag
12	`error_status`	Status (10.2)	Node → Server	Error status message
Field Registration				
20	`telemetryValue_registration`	FieldRegistration (10.3)	Node → Server	Register a telemetryVa
21	`parameter_registration`	FieldRegistration (10.3)	Node → Server	Register a parameter fi
Telemetry Group Management				
30	`telemetry_group_definition`	TelemetryGroupDefinition (10.4)	Node → Server	Define a group of fiel updates
31	`telemetry_group_update`	TelemetryGroupUpdate (10.5)	Node → Server	Update values for a fiel
Parameter Management				
50	`parameter_set_req`	ParameterSetReq (10.7)	Server/Node → Node	Request to set a param
51	`parameter_set_confirmation`	ParameterSetConfirmation (10.10)	Node → Server/Node	Response with confirme value
52	`parameter_set_lock_req`	ParameterSetLock (10.15)	Server/Node → Node	Request to lock a para
53	`parameter_set_lock_confirmation`	~~ParameterSetLock (10.15~~ParameterSetLockConfirmation (10.16)	Node → Server	Response confirming pa
Field Access				
60	`field_get_req`	FieldGetReq (10.11)	Server/Node → Node	Request field value
61	`field_get_res`	FieldGetRes (10.12)	Node → Server/Node	Response with field val
62	`field_id_lookup_req`	FieldIDLookupReq (10.13)	Node → Node	Requests the fieldID of
63	`field_id_lookup_res`	FieldIDLookupRes (10.14)	Node → Node	Response with field ID
Heartbeat				

Enum	Message Type	Data Payload	Direction	Description
40	`heartbeat_req`	HeartBeat (10.6)	Server → Node	Heartbeat request
41	`heartbeat_res`	HeartBeat (10.6)	Node → Server	Heartbeat response

10 Data Structures

10.1 NodeInfoRes

Response containing information about a node's capabilities.

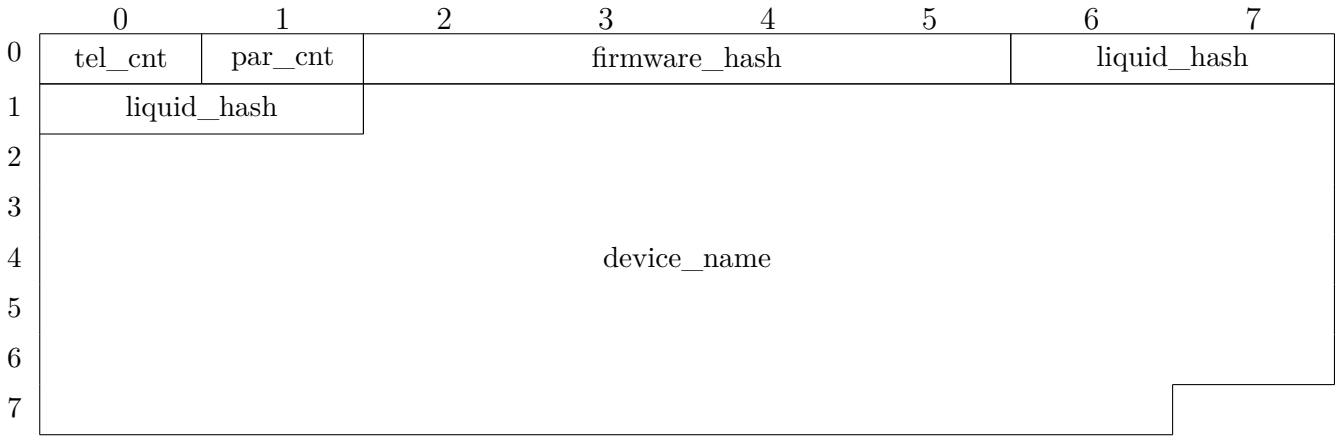

Figure 2: NodeInfoRes byte layout (8 bytes per row)

Field Descriptions:

Field	Bytes	Description
tel_cnt	1	Number of telemetryValues on this node
par_cnt	1	Number of parameters on this node
firmware_hash	4	Hash of the firmware version
liquid_hash	4	Hash of the LiquidCan protocol version
device_name	≤53	Human-readable device name (null-terminated)
Total	**11–63**	

```
typedef struct __attribute__((packed)) {
    uint8_t var_count;
    uint8_t par_count;
    uint32_t firmware_hash;
    uint32_t liquid_hash;
    char device_name[53];
} node_info_res_t;
```

Listing 1: NodeInfoRes struct

10.2 Status

General status message with text information.

Field	Bytes	Description
msg	≤63	Status message text (null-terminated)

```
typedef struct __attribute__((packed)) {
    char msg[63];
} status_t;
```

Listing 2: Status struct

10.3 FieldRegistration

Registration information for a telemetryValue or parameter field. The DataType here refers to the DataType Enum value (see 10.17).

Field	Bytes	Description
field_id	1	Unique identifier for this field
field_type	1	Data type (DataType enum)
field_name	≤61	Human-readable field name (null-terminated)
Total	**3–63**	

```
typedef struct __attribute__((packed)) {
    uint8_t field_id;
    uint8_t field_type;
    char field_name[61];
} field_registration_t;
```

Listing 3: FieldRegistration struct

10.4 TelemetryGroupDefinition

Defines a group of related telemetryValues for efficient batch updates.

Field	Bytes	Description
group_id	1	Unique identifier for this group
field_ids	≤62	Array of field IDs in this group (terminated by end of frame)
Total	**2–63**	

```
typedef struct __attribute__((packed)) {
    uint8_t group_id;
    uint8_t field_ids[62];
} telemetry_group_definition_t;
```

Listing 4: TelemetryGroupDefinition struct

10.5 TelemetryGroupUpdate

Updates all telemetryValues from a previously defined group.

Field	Bytes	Description
group_id	1	Group identifier
values	≤62	Packed values of all telemetry values in the group
Total	**2–63**	

Note: The values are packed in the same order as announced in the TelemetryGroupDefinition.

```
typedef struct __attribute__((packed)) {
    uint8_t group_id;
    uint8_t values[62];
} telemetry_group_update_t;
```

Listing 5: TelemetryGroupUpdate struct

10.6 HeartBeat

Heartbeat message with counter.

Field	Bytes	Description
counter	4	Incrementing counter value

```c
typedef struct __attribute__((packed)) {
    uint32_t counter;
} heartbeat_t;
```

Listing 6: HeartBeat struct

10.7 ParameterSetReq

Request to set a parameter value.

Field	Bytes	Description
parameter_id	1	Parameter identifier
value	~~61~~ 1–4	New value (~~type~~ size depends on parameter datatype)
Total	**~~62~~2–5**	

```c
typedef struct __attribute__((packed)) {
    uint8_t parameter_id;
    ~~uint8_t value[61];~~
    uint8_t value[]; // 1-4 bytes, flexible array member
} parameter_set_req_t;
```

Listing 7: ParameterSetReq struct

10.8 FieldStatus

Status codes for field access operations:

Enum Value	Status Name	Description
0	`Ok`	Field exists and operation was successful
1	`NotFound`	Field does not exist

10.9 ParameterSetStatus

Status codes for parameter set operations:

Enum Value	Status Name	Description
0	`Success`	Parameter was successfully set
1	`InvalidParameterID`	The parameter ID does not exist
2	`ParameterLocked`	The parameter is locked and cannot be modified
3	`NodeToNodeModification`	The parameter was modified by another node

10.10 ParameterSetConfirmation

Response to a parameter set request.

Field	Bytes	Description
parameter_id	1	Parameter identifier
status	1	Status code (ParameterSetStatus enum)
value	~~61~~ 1–4	Confirmed value after set operation (size depends on datatype)
Total	**~~63~~3–6**	

Note: If the parameter is not found (field_status = NotFound), the value field should contain the originally requested value from the ParameterSetReq message, and the status field should indicate InvalidParameterID.

```
typedef struct __attribute__((packed)) {
    uint8_t parameter_id;
    uint8_t status;
    ~~uint8_t value[61];~~
    uint8_t value[]; // 1-4 bytes, flexible array member
} parameter_set_confirmation_t;
```

Listing 8: ParameterSetConfirmation struct

10.11 FieldGetReq

Request to retrieve a field value

Field	Bytes	Description
field_id	1	Field identifier

```
typedef struct __attribute__((packed)) {
    uint8_t field_id;
} field_get_req_t;
```

Listing 9: FieldGetReq struct

10.12 FieldGetRes

Response with requested field value.

Field	Bytes	Description
field_id	1	Field identifier
field_status	1	Field existence status (FieldStatus enum)
value	~~61~~ 1–4	Field value (size depends on field datatype)
Total	**~~63~~3–6**	

Note: If the field is not found (field_status = NotFound), the field_id should still contain the requested field ID from the FieldGetReq message, and the value field content is undefined.

```
typedef struct __attribute__((packed)) {
    uint8_t field_id;
    uint8_t field_status;
    ~~uint8_t value[61];~~
    uint8_t value[]; // 1-4 bytes, flexible array member
} field_get_res_t;
```

Listing 10: FieldGetRes struct

10.13 FieldIDLookupReq

Requsts the fieldID of a field matching a field name

Field	Bytes	Description
field_name	≤61	Field Name (null-terminated)
Total	**2–61**	

```
typedef struct __attribute__((packed)) {
    uint8_t field_name[61];
} field_id_lookup_req_t;
```

Listing 11: FieldIDLookupReq struct

10.14 FieldIDLookupRes

Response with requested field id and the datatype of the field.

Field	Bytes	Description
field_id	1	Field ID
field_type	1	Field Datatype
field_status	1	Field existence status (FieldStatus enum)
Total	**3**	

Note: If the field is not found (field_status = NotFound), the field_id should be set to 0 (reserved invalid ID) and field_type is undefined.

```
typedef struct __attribute__((packed)) {
    uint8_t fieldID;
    uint8_t field_type;
    uint8_t field_status;
} field_id_lookup_res_t;
```

Listing 12: FieldIDLookupRes struct

10.15 ParameterSetLock

Locks a parameter to prevent changes.

Field	Bytes	Description
parameter_id	1	Parameter identifier to lock
parameter_lock	1	Lock status(0=unlocked, 1=locked)
Total	**2**	

```
typedef struct __attribute__((packed)) {
    uint8_t parameter_id;
    uint8_t lock_status;
} parameter_set_lock_t;
```

Listing 13: ParameterSetLock struct

10.16 ParameterSetLockConfirmation

Confirmation of a parameter locking request.

Field	Bytes	Description
parameter_id	1	Parameter identifier to lock
parameter_lock	1	Lock status(0=unlocked, 1=locked)
field_status	1	Field status (FieldStatus enum)
Total	**3**	

Note: If the parameter is not found (field_status = NotFound), the parameter_id should still contain the requested parameter ID from the ParameterSetLockReq message.

```
typedef struct __attribute__((packed)) {
    uint8_t parameter_id;
    uint8_t lock_status;
    uint8_t field_status;
} parameter_set_lock__confirmation_t;
```

Listing 14: ParameterSetLockConfirmation struct

10.17 DataType

The protocol supports the following data types:

Enum Values	Type Name	Description
0	`Float32`	32-bit floating point
1	`Int32`	32-bit signed integer
2	`Int16`	16-bit signed integer
3	`Int8`	8-bit signed integer
4	`UInt32`	32-bit unsigned integer
5	`UInt16`	16-bit unsigned integer
6	`UInt8`	8-bit unsigned integer
7	`Boolean`	8-bit value: non 0 value equals true

11 Versioning and Extension Mechanisms

This protocol uses semantic versioning. See https://semver.org/ for a detailed description. A minor update version would, for example, be adding a new datatype or a new message type . Every update of the document must trigger an update of the liquidCAN repo, containing the Rust and C code for each implementation. The updated repo must be reflected in the `liquidcan_hash` field of the `node_info_announcement` used in the firmware/server.

The Firmware in the system can run with minor or patch version differences, "major" changes are not allowed to run simulaneously. The server can inform the operators in case there is firmware running with different major versions.