LiquidCan Protocol Documentation

for Distributed Embedded Systems
at the TU Wien Space Team

TU Wien Space Team

Version 0.1

December 16, 2025

Version History

Version	Date	Changes	Author
0.1	2025-12-12	Initial protocol specification	Raffael Rott

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 the way our client devices (such as ECUs) communicate with the central server and with each other. It is designed to be as simple and extensible as possible. Care has been taken to minimize the amount of common type definitions between the server and the nodes.

The goal of this document is not to explain the system architecture but to describe how a multi-node system can interact through a CAN bus with a central server.

2 Notation and Conventions

- This protocol uses the CAN-FD extension
- All fields are little endian.
- Payload length: 64 Byte CAN FD.

2.1 Common Terms

Term	Description
(CAN) Client	A CAN client is a device which is connected to the bus
Variable	A variable is a non-externally modifiable value which gets periodically sent to the server
Parameters	Parameters can be externally modified
Field	An encompassing term for variables 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 of 64 bytes. The first byte of each message contains 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	63	Payload data

5 Node Registration

As soon as a node comes online (or when it receives a `node_info_req`, see 2), it sends out a `node_info_res` (see 2). This announces the node to the bus and includes its name, number of variables 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 variables and parameters. Variables 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 variables, modifiable by either the server or other nodes.

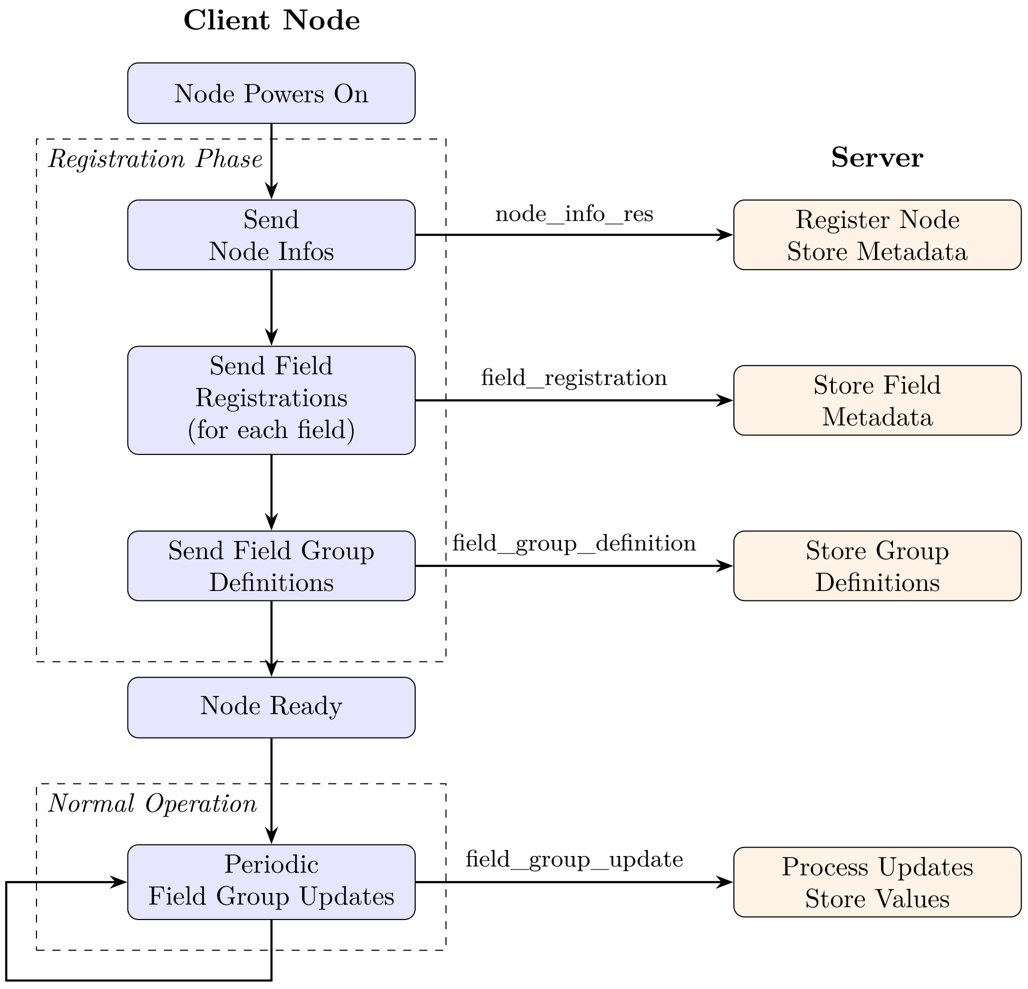

Figure 1: Field Registration and Update Flow

6.1 Registration

Variables and parameters are dynamically defined over the bus.

After initializing registration, a node sends out `field_registration` messages, one for each parameter/variable. The FieldRegistration includes a per-node-unique field ID (parameters and variables can have the same ID on the same node), the datatype of the field, and a human-readable name. From this point on, the server knows which types of fields the node has.

Next, the node sends `field_group_definition` messages. Variables and parameters cannot be mixed together in one FieldGroupDefinition. . 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 `field_group_updates`. The node must ensure that all of the fields defined can fit into a FieldGroupUpdate.

6.2 Regular Operation

6.2.1 Field Updates

During regular operation, each node sends `field_group_update` messages at a defined interval. The interval can vary between groups, allowing nodes to send fields 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 changes the internal parameter value and responds with a `parameter_set_res` message containing the new value. This should be the actual value read back from the parameter, 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_res` message to the server.

6.2.3 Parameter Locking

A parameter can optionally be locked and later unlocked through a `parameter_set_lock_req` message (see 10.11 and 2). 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. **TODO:** Is this needed as part of the protocol?

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 and the value of the field.

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	Description
Node Discovery and Information			
0	`node_info_req`	No payload	Request node information
1	`node_info_res`	NodeInfoRes (10.1)	Response with node capabilities and identification
Status Messages			
10	`info_status`	Status (10.2)	Informational status message
11	`warning_status`	Status (10.2)	Warning status message
12	`error_status`	Status (10.2)	Error status message
Field Registration			
20	`variable_registration`	FieldRegistration (10.3)	Register a variable field
21	`parameter_registration`	FieldRegistration (10.3)	Register a parameter field
Field Group Management			
30	`field_group_definition`	FieldGroupDefinition (10.4)	Define a group of fields for batch updates
31	`field_group_update`	FieldGroupUpdate (10.5)	Update values for a field group
Heartbeat			
40	`heartbeat_req`	HeartBeat (10.6)	Heartbeat request
41	`heartbeat_res`	HeartBeat (10.6)	Heartbeat response
Parameter Management			
50	`parameter_set_req`	ParameterSetReq (10.7)	Request to set a parameter value
51	`parameter_set_res`	ParameterSetRes (10.8)	Response with confirmed parameter value
52	`parameter_set_lock_req`	ParameterSetLock (10.11)	Request to lock a parameter
53	`parameter_set_lock_res`	ParameterSetLock (10.11)	Response confirming parameter lock
Field Access			
60	`field_get_req`	FieldGetReq (10.9)	Request field value
61	`field_get_res`	FieldGetRes (10.10)	Response with field value

10 Data Structures

10.1 NodeInfoRes

Response containing information about a node's capabilities.

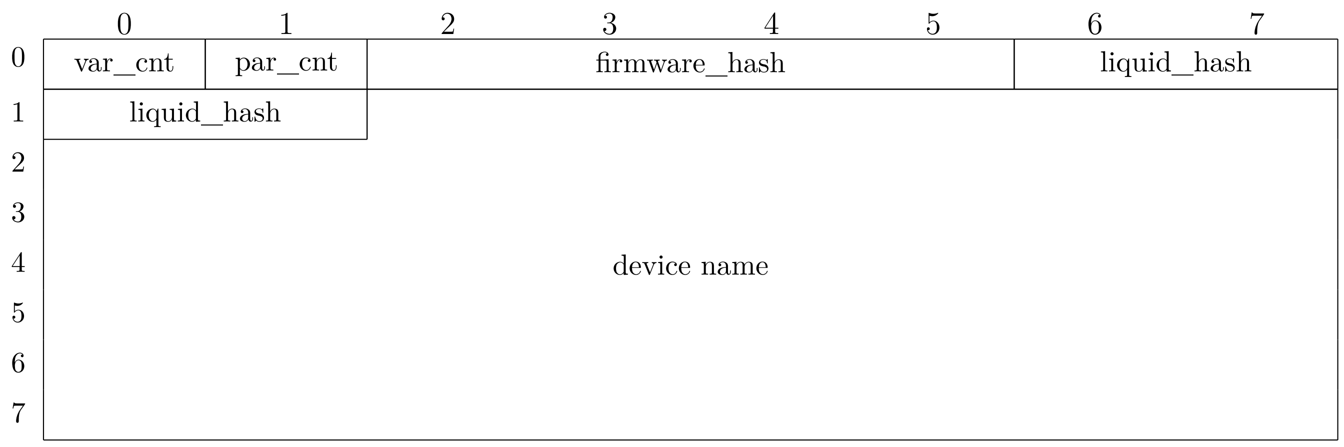

Figure 2: NodeInfoRes byte layout (8 bytes per row)

Field Descriptions:

Field	Bytes	Description
variable_count	1	Number of variables on this node
parameter_count	1	Number of parameters on this node
firmware_hash	4	Hash of the firmware version
liquidcan_hash	4	Hash of the LiquidCan protocol version
device_name	53	Human-readable device name
Total	**63**	

```c
typedef struct __attribute__((packed)) {
    uint8_t variable_count;
    uint8_t parameter_count;
    uint32_t firmware_hash;
    uint32_t liquidcan_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

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

Listing 2: Status struct

10.3 FieldRegistration

Registration information for a variable or parameter field. The DataType here refers to the DataType Enum value (see 10.12).

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
Total	**63**	

```c
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 FieldGroupDefinition

Defines a group of related fields for efficient batch updates. The

Field	Bytes	Description
group_id	1	Unique identifier for this group
field_ids	62	Array of field IDs in this group
Total	**63**	

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

Listing 4: FieldGroupDefinition struct

10.5 FieldGroupUpdate

Updates all field values in a previously defined group.

Field	Bytes	Description
group_id	1	Group identifier
values	62	Packed values for all fields in the group
Total	**63**	

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

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

Listing 5: FieldGroupUpdate struct

10.6 HeartBeat

Simple 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	62	New value (type depends on parameter)
Total	**63**	

```c
typedef struct __attribute__((packed)) {
    uint8_t parameter_id;
    uint8_t value[62];
} parameter_set_req_t;
```

Listing 7: ParameterSetReq struct

10.8 ParameterSetRes

Response to a parameter set request.

Field	Bytes	Description
parameter_id	1	Parameter identifier
value	62	Confirmed value after set operation
Total	**63**	

```c
typedef struct __attribute__((packed)) {
    uint8_t parameter_id;
    uint8_t value[62];
} parameter_set_res_t;
```

Listing 8: ParameterSetRes struct

10.9 FieldGetReq

Request to retrieve a field value. Parameters are symbolized by field_type = 0 and variables by field_type = 1.

Field	Bits/Bytes	Description
field_type	1 bit	Type of field(parameter or variable)
field_id	1 Byte	Field identifier

```c
typedef struct __attribute__((packed)) {
    uint8_t field_id : 1;
    uint8_t field_id : 8;
} field_get_req_t;
```

Listing 9: FieldGetReq struct

10.10 FieldGetRes

Response with requested field value.

Field	Bytes	Description
field_id	1	Field identifier
value	62	Field value
Total	**63**	

```c
typedef struct __attribute__((packed)) {
    uint8_t field_id;
    uint8_t value[62];
} field_get_res_t;
```

Listing 10: FieldGetRes struct

10.11 ParameterSetLock

Locks a parameter to prevent changes.

Field	Bytes	Description
parameter_id	1	Parameter identifier to lock

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

Listing 11: ParameterSetLock struct

10.12 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

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_res` used in the firmware/server.