Niklavs Geizans

Modular CAN Bus based control unit

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Degree Programme in Electronics

Bachelor’s Thesis

April 2017

Author Title Number of Pages Date	Niklavs Geizans Modular CAN Bus based control unit 27 pages + 3 appendices 24 April 2017
Degree	Bachelor of Engineering
Degree Programme	Electronics
Specialisation option	
Instructor(s)	Janne Mäntykoski, Senior Lecturer

This Final Year Project (FYP) was carried out for a Finnish based research and development company – Janesko Oy. The goal was to design a modular and highly customisable control unit for specialized optical sensor (refractometer) control and monitoring.

Design was based on an existing implementation called a Multichannel User Interface (MI). Two modularity limiting factors in MI were identified and designs to address them were proposed and tested.

Firstly, over-current protection circuitry was re-implemented to not require a microcontroller assisted function. The new circuitry was simulated in LT Spice IV simulation software and confirmed to be functioning. Secondly, an extension to a proprietary communications protocol called MTR CAN was developed in order to address its modularity limiting factors. The new extension was tested on the existing MI electronics and confirmed to be functioning.

While a prototype implementation of the new control system could not be done due to time constrains, the proposed design has been shown by testing and simulation to be viable and with further development has potential in a future commercial application.

Keywords	Refractometers, Modular, Industrial, CAN, Controller Area Network, Process Control.

Contents

1 Introduction 1

2 Theoretical Background 2

2.1 Multichannel User Interface 2

2.1.1 MI Overview 2

2.1.2 Industrial PC 3

2.1.3 MI Motherboard 3

2.1.4 Plug-in Modules 3

2.2 New Proposed Design 4

2.3 Introduction of CAN bus 5

2.3.1 Technical overview 6

2.3.2 Bus Values 7

2.3.3 Bus network topology 8

2.3.4 Message Frames 9

2.3.5 Bit Arbitration and message collision avoidance 10

3 Implementation 12

3.1 Current protection 12

3.2 Controller Area Network (CAN) 19

3.2.1 Existing Application Layer Protocol 19

3.2.2 Extended Application Layer Protocol 20

3.2.3 Dynamic node identifier distribution 22

4 Conclusions 26

References 27

Appendices

Appendix 1. Over-current protection circuit simulation schematic (LT Spice IV)

Appendix 2. Over-current protection circuit simulation graphs (LT Spice IV)

Appendix 3. Dynamic node ID distribution code

1 Introduction

This thesis was done in cooperation with Finnish based research and development company Janesko Oy. Janesko primarily designs sensors for solution concentration measurements called refractometers. Refractometers are optical process control instruments that are mounted within a pipe system of customer’s industrial or manufacturing equipment.

Multiple refractometers and other additional measurement instruments can be used in conjunction to provide more advanced data acquisition and/or functionality than is possible using only a single refractometer. Example of such functionality would be the refractometer’s optical element washing. Implementation of this control functionality and any additional features requires a centralized control unit.

Because the end-users’ needs depend on case to case bases, and the product installations are tailored for each end-user, the aforementioned control unit should be highly customizable. Additional features would have to be possible to add with a software update and/or additional extension hardware.

This introduces the need for the control unit to have an expandable architecture. Modular implementation approach can fulfil this need, therefore, the scope of this thesis work is to create a prototype design of a fully modular control unit as a proof of concept for a possible further product development.

2 Theoretical Background

2.1 Multichannel User Interface

Design practices at Janesko Oy highly encourage design reuse to ensure rapid design process and minimize work time. Due to this reason, the work is done on the back of an existing Janesko design and implementation. The final design this thesis aims to provide is based on a design called Multichannel User Interface (MI).

2.1.1 MI Overview

MI is a centralized control unit designed to collect and process refractometer data, and provide a way for a technician to interface with refractometer installation [1,1]. It supports up to 4 ethernet interfaced refractometers, and additionally up to 8 plug-in module cards. A 3D representation of MI is shown in Figure 1.

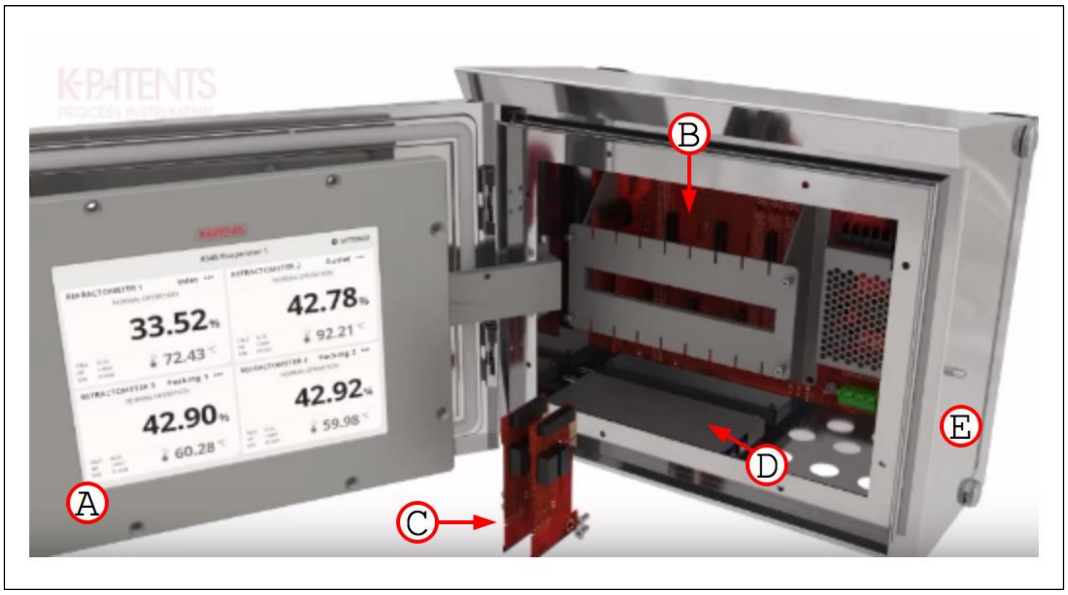


Figure 1. Rendering of MI (A. industrial PC, B. motherboard, C. plug-in module card, D. ethernet switch, E. enclosure). (Copied video frame from K-Patents Multichannel User Interface MI video. K-Patents; 2016. URL: https://www.youtube.com/watch?v=fad0mAE0Yjk) [2]

MI components can be broken down into 4 main component groups: industrial PC with display, motherboard, ethernet switch, and plug-in modules. The MI base configuration always contains at least the industrial PC, motherboard, and an ethernet switch.

2.1.2 Industrial PC

Industrial PC is the core computing part of the MI. It is responsible for collection and processing of data. Additionally, the built-in touchscreen display allows for refractometer data to be displayed in a human-readable manner.

2.1.3 MI Motherboard

Motherboard has two main functions. Firstly, it is responsible for power distribution as it contains power supply and an over-current protection measures for every device it powers. Secondly, it provides slots for MI plug-in modules, and the communication network (CAN bus) connecting them.

2.1.4 Plug-in Modules

MI plug-in modules are expansion cards that can be plugged into motherboard to give additional functionality not present in basic MI configuration. As of writing of this paper there are currently four module types officially supported.

1. General Purpose Input Output (GPIO).
2. Milliampere Output.
3. Milliampere Input.
4. Relay.

2.2 New Proposed Design

The physical form-factor and the motherboard are identified as the limiting factors of modularity in the MI. This is because the motherboard is limiting the number of expansion modules that can be used. To address this issue a new form-factor is proposed, which would not require use of a motherboard. For clarity, the new proposed design is referred to as Modular Control System (MCS) onwards.

This is achieved by moving the industrial PC, plug-in modules, and the ethernet switch to separate enclosures. These enclosures would be mounted on a DIN rail. As it is a common industry standard mounting mechanism. See Figure 2.

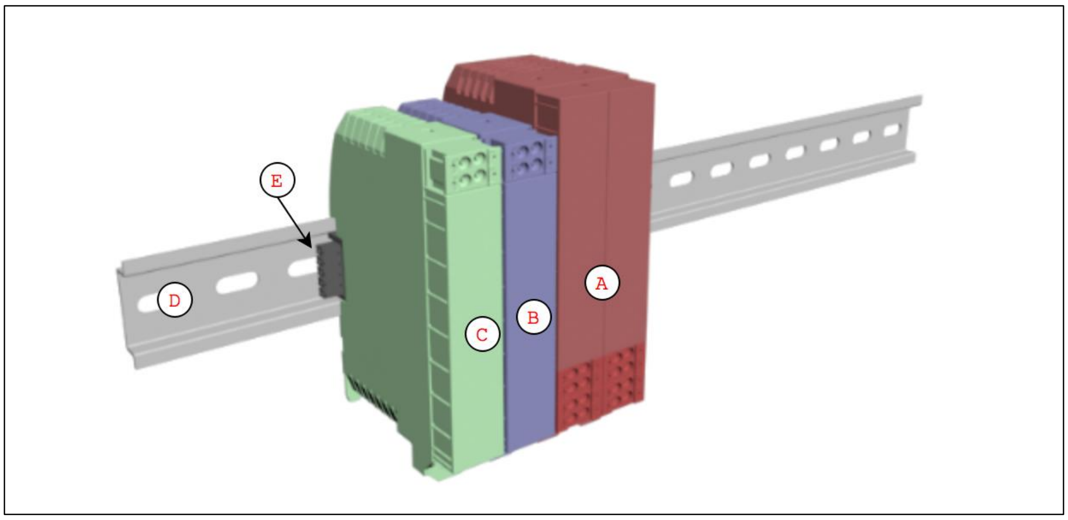


Figure 2. Rendering of MCS mounted onto DIN rail (A. master module, B and C. Slave modules, D. DIN rail, E. inter-module bus connection)

Removal of the motherboard brings forth a set of challenges. Addressing them is paramount to make this a viable design. Firstly, motherboard in the MI contains the physical part of the data communication network between the MI's components. Secondly, the over-current protection electronics both for plug-in modules and the industrial PC are located on the motherboard. This means that the over-current protection must be redesigned to not require motherboard. And lastly, expansion modules use motherboard plug-in slots to obtain an identifier. CAN network communication requires this identifier to work properly, therefore a new identification scheme must be devised.

2.3 Introduction of CAN bus

Controller Area Network (CAN) is an automotive communications bus protocol developed by Robert Bosch GmbH. Due to rising issue of growing wiring complexity in vehicles in the early 1980s Bosch started to consider the use of serial bus communication to ensure connectivity between vehicle components [3]. At the time, no other serial bus protocol sufficiently fulfilled the requirements for safety critical automotive applications, therefore, a team at Bosch started the development of a new vehicle serial bus that is now known as CAN bus [3].

The first version of CAN specification was publicly revealed in 1986 at SAE (Society of Automotive Engineers) congress held in Detroit. Five years later in September of 1991 the CAN specification 2.0 was published by Bosch [3]. CAN 2.0 is the latest version of specification. It is split into two parts: part A (description of standard message format) and part B (description of an extended message format) [4].

While CAN bus was designed for primarily automotive use in passenger cars, it was adopted early on by other industries such as Industrial, aerospace, and even medical fields. The widespread adoption of CAN was promoted by its built-in error detection, message collision avoidance, and multi-master architecture.

2.3.1 Technical overview

Within the Open Systems Interconnection (OSI) model CAN bus defines the Data Link Layer and a part of the Physical Layer. As shown in Figure 3, CAN specification 2.0 defines its own abstraction layers that overlap with OSI model, but do not fully follow it. [4]

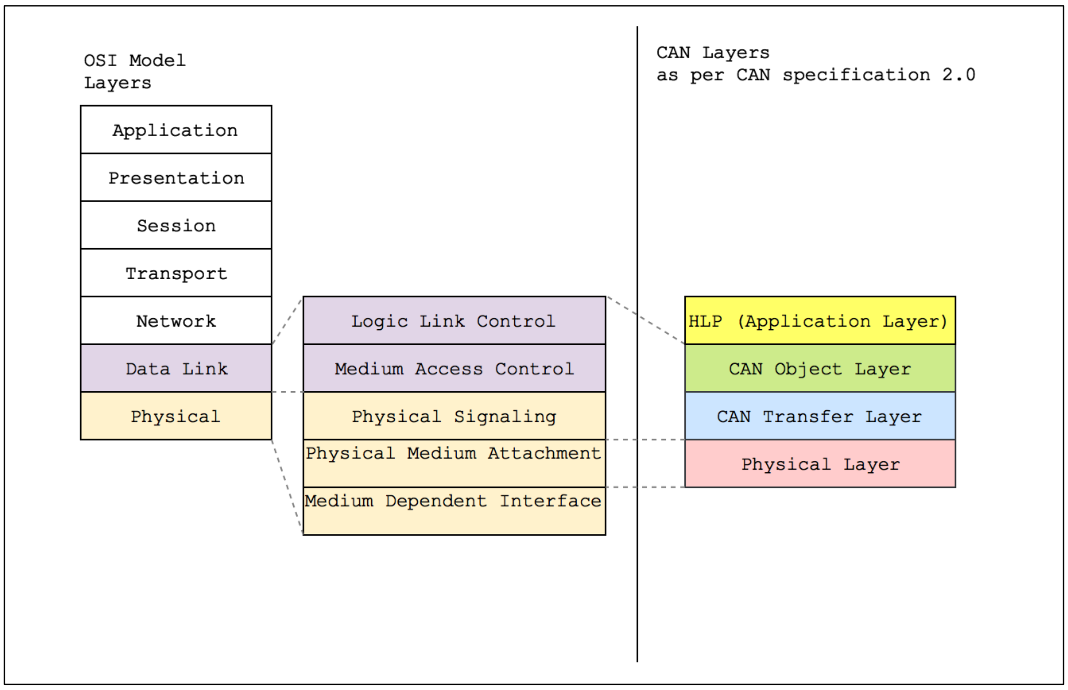


Figure 3. OSI model layers in respect to CAN specification defined abstraction layers.

CAN specification does not define the implementation of Network Layer and higher layers, therefore, their implementation is left up to the network designer. Protocols implementing these OSI Model layers are called Higher Layer Protocols (HLPs). Alternatively, one of the publicly available HLPs such as CANopen, UAVCAN, or CanKingdom can be used. [5]

Both CAN Object and CAN Transfer Layers are typically implemented using integrated chips designed for exactly this purpose called CAN Controllers. It is either wired between CAN Transceiver and microcontroller, or integrated inside the microcontroller.

CAN Transceiver is responsible for converting signals from CAN Controller to be applicable for transmission on the network. See Figure 4.

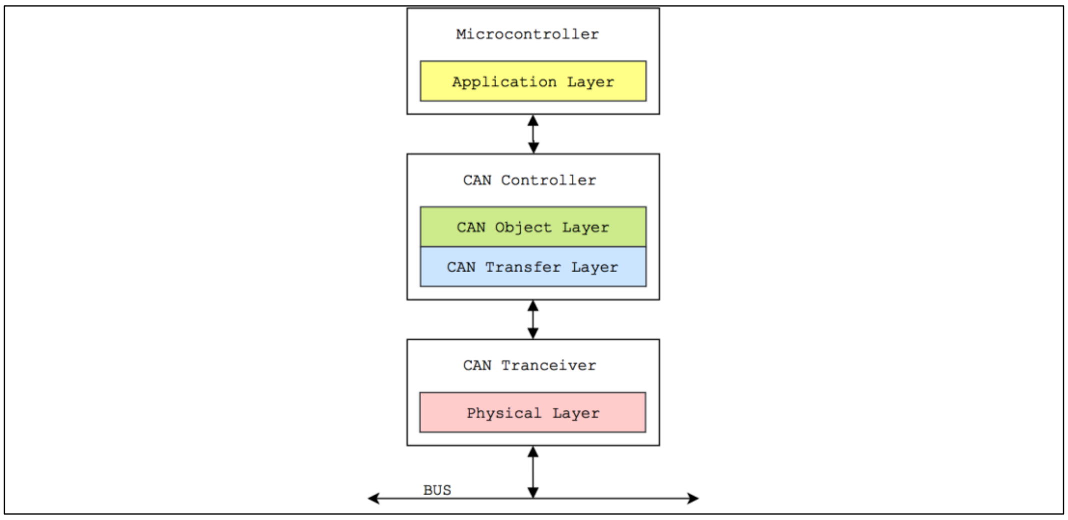


Figure 4. Components responsible for CAN Layer implementation.

Typically, the transmission medium is a twisted wire pair with terminating resistance of 60 ohms between them [6,7].

2.3.2 Bus Values

The logic voltage levels on transmission medium are not specified by CAN specification, however, two logic states and their behaviour is defined: recessive and dominant [4]. See Figure 5.

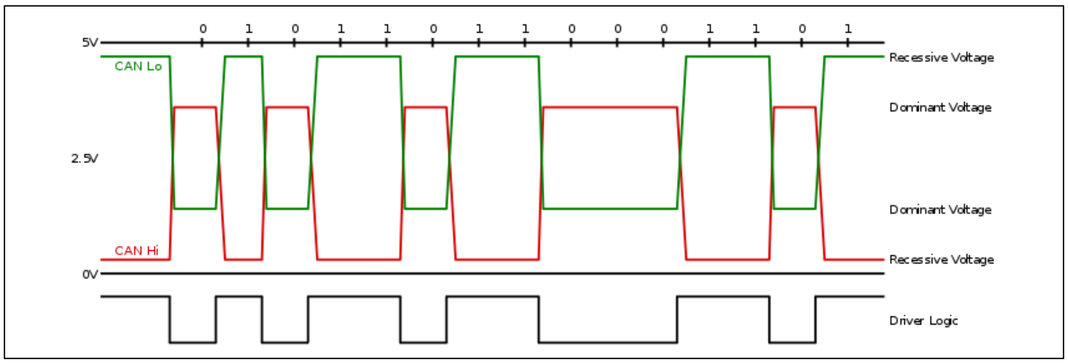


Figure 5. Dominant and Recessive states. (Copied from Wikipedia article on CAN bus. URL: https://en.wikipedia.org/wiki/CAN_bus) [5]

Recessive state is considered when differential voltage between bus wires is high, and dominant state is considered when differential voltage between the lines is low. If two nodes are trying to drive the bus to two different states, the dominant state is seen on the bus. [6]

2.3.3 Bus network topology

One of the defining features of CAN is its bus type physical topology. This means that all devices (nodes) are interconnected in the network using a single cable (backbone), as shown in Figure 6. Stubs are used to connect each node to the backbone. [7]

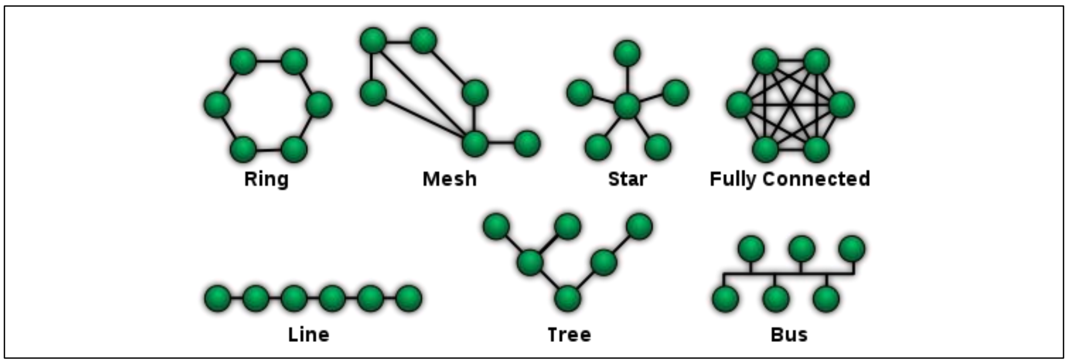


Figure 6. Seven common network topologies. Starting from top left: ring, mesh, star, fully connected, line, tree, and bus. (Copied from Wikipedia article on network topologies, URL: https://en.wikipedia.org/wiki/Network_topology) [7]

This type of network topology brings both several advantages and disadvantages. Main advantage is the reduction of wiring necessary to interconnect all of the nodes on the bus. Additional benefit is the expandability gained, as the bus allows for additional nodes to be added to the network without having to rewire it. Unfortunately, three major disadvantages can be identified.

Firstly, the network can have only a single message transmitted at any given time. What’s more, because every node can start transmitting a message at any given time, a possibility of a message collision and subsequent data loss can occur. This is addressed by a rigidly defined bit transmission sequences (message frames), and a bit-arbitration mechanism.

Secondly, data transmitted on the bus is seen by all connected nodes. While this is useful when global broadcasts of information are necessary, it becomes a problem when data has to be transmitted to a specific node on the network. On CAN bus this issue is addressed by attaching a message identifier to every data packet transmitted.

And lastly, the bus network is susceptible to global failure. If a physical issue on the backbone occurs, or a node is misbehaving, the whole network can be brought down [7]. While this drawback is inherent within bus topology, a catastrophic system failure can be avoided using error detection and signalling scheme.

2.3.4 Message Frames

CAN bus uses rigidly defined message frames for data transfer, data requests, and error state signalling. Four types of frames are defined: data, remote, error, and overload [4,10]. This paper will focus on data frame specifically; as remote frames are not used in this application, and overload and error frames are fully managed by CAN Controller.

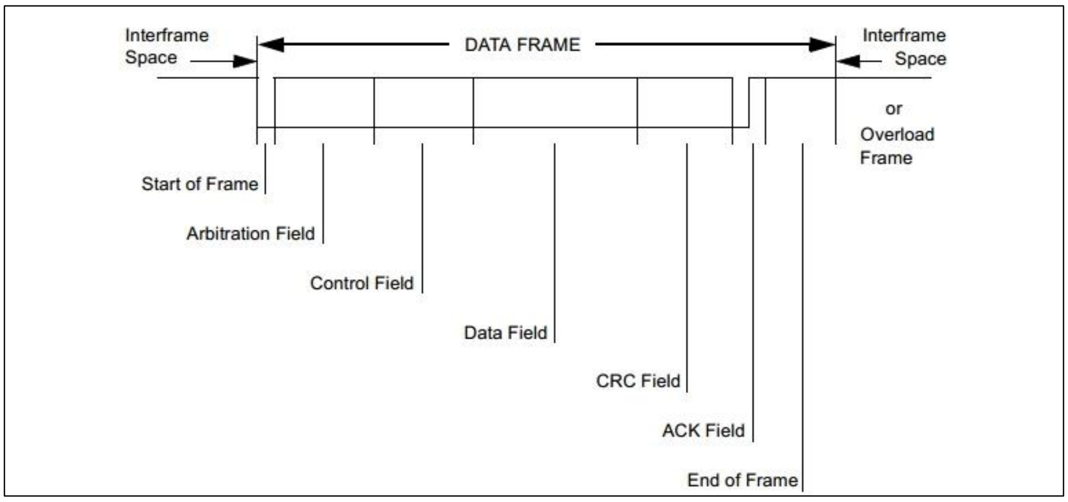


Figure 7. CAN bus data frame and its bit-fields (Copied from CAN Specification, Version 2.0. Robert Bosch GmbH; 1991). [4]

As shown in Figure 7, data frame contains seven bit-fields: Start of Frame (SOF), arbitration field, control field, data field, cyclic redundancy check (CRC) field, acknowledge (ACK) field, and End of Frame (EOF). [4,10]

2.3.5 Bit Arbitration and message collision avoidance

To avoid message collision and data loss caused by it, CAN bus implements a set of strict rules for when a node is allowed to send a message frame on the bus. Nodes are only permitted to start transmission of data and remote frames if inter-frame space is detected on the bus [4,43]. Inter-frame space means that the bus is in idle state. While this ensures that a node will not start transmitting a frame while a message/request frame from another node is already being sent, it will not prevent situation where more than one node transmits a message frame at the same time. This is addressed by a bit-arbitration mechanism.

CAN Specification 2.0 defines two message frame formats for data and remote frames: standard and extended. The difference between these formats is the arbitration field. Extended format has an arbitration field with width of 32 bits (29-bit combined identifier), whereas, standard frame has 12 bit (11-bit identifier) wide arbitration field. Bit arbitration applies only to arbitration field of data and remote frames. See contents of arbitration field in Figure 8. [4]

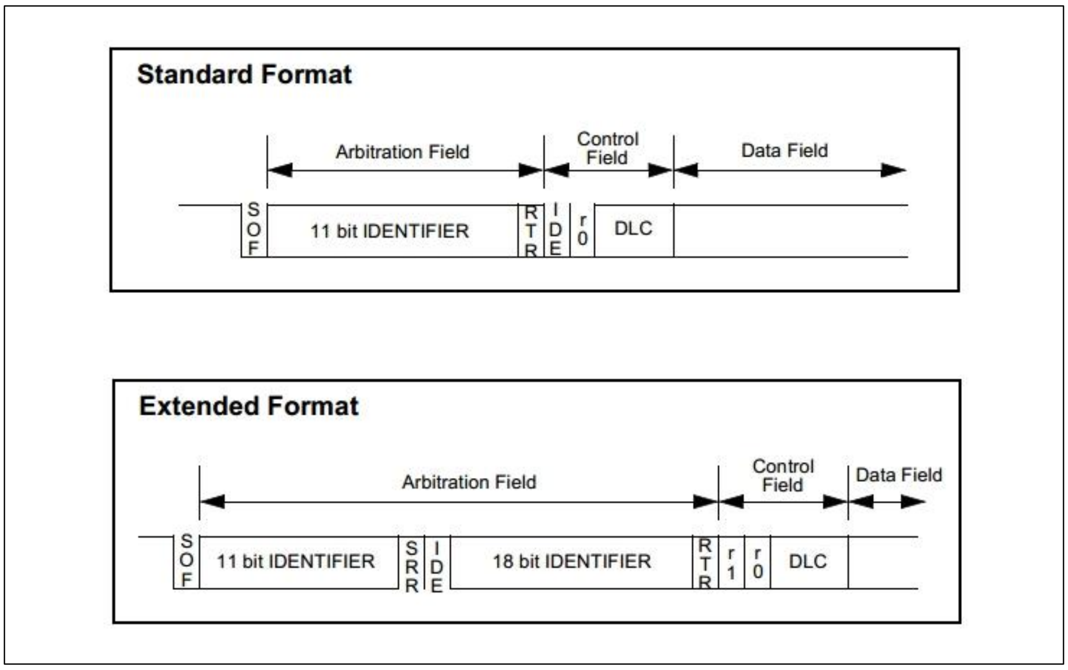
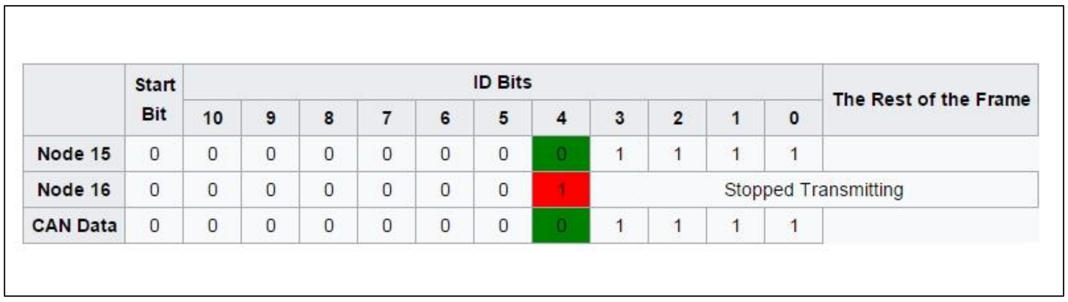


Figure 8. Arbitration field and its bit contents (Copied from CAN Specification, Version 2.0. Robert Bosch GmbH; 1993). [4]

When two or more nodes begin transmission of a frame at the same time, the first non-inter-frame space bit seen on the bus is SOF (always dominant), followed by arbitration field. SOF bit signals to all the nodes that a message is being transmitted. During the transmission of arbitration field, all nodes read the bus state. [4]

If any of the transmitting nodes detect that a bit on the bus that does not match the bit they are transmitting, then the node aborts transmission and waits for the next time bus is free i.e. inter-frame space present on the bus. See Figure 9. [4]



	Start Bit	ID Bits											The Rest of the Frame
		10	9	8	7	6	5	4	3	2	1	0	
Node 15	0	0	0	0	0	0	0	0	1	1	1	1	
Node 16	0	0	0	0	0	0	0	1	Stopped Transmitting				
CAN Data	0	0	0	0	0	0	0	0	1	1	1	1	

Figure 9. Visual representation of bit arbitration process. (Copied from Wikipedia article on CAN bus. URL: https://en.wikipedia.org/wiki/CAN_bus) [5]

At the end of arbitration field only the node with highest priority ID (Lowest Identifier number) will continue transmission of its frame. This node is said to have won the bit-arbitration.

3 Implementation

3.1 Current protection

Over-current protection in MI is designed so that the motherboard monitors the current draw of the plug-in modules and the Industrial PC. A dedicated current measurement block is present for each device connected to the motherboard.

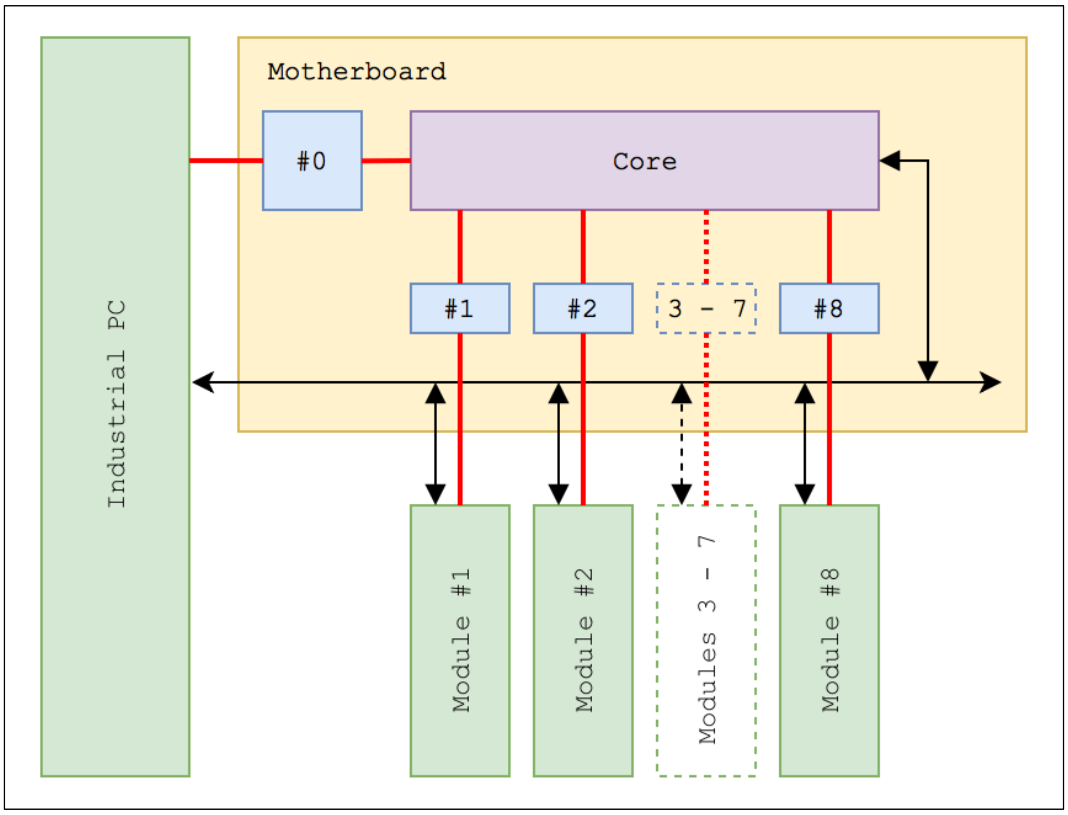


Figure 10. Simplified MI block diagram (green – module/industrial PC, blue – current measurement block, red – power supply bus, black – CAN bus, yellow - motherboard).

Current measurement block in MI is implemented using a sense resistor. Voltage drop across this resistor is amplified, converted to digital data using an ADC, and current value is calculated from amplified voltage drop in motherboard's CPU. If the actual current consumption is greater than the programmed maximum limit, motherboard de-energizes a MOSFET switch which supplies power to module or industrial PC. See Figure 10.

In MCS the current measurement block is moved to modules, as shown in Figure 11. This means that a central supervisory microcontroller cannot monitor current draw anymore. Since the current protection requires ability to remove power from module microcontroller, it cannot be a part of the current protection mechanism itself. Therefore, for MCS modules a new current protection mechanism must be implemented which would work standalone from the module microcontroller.

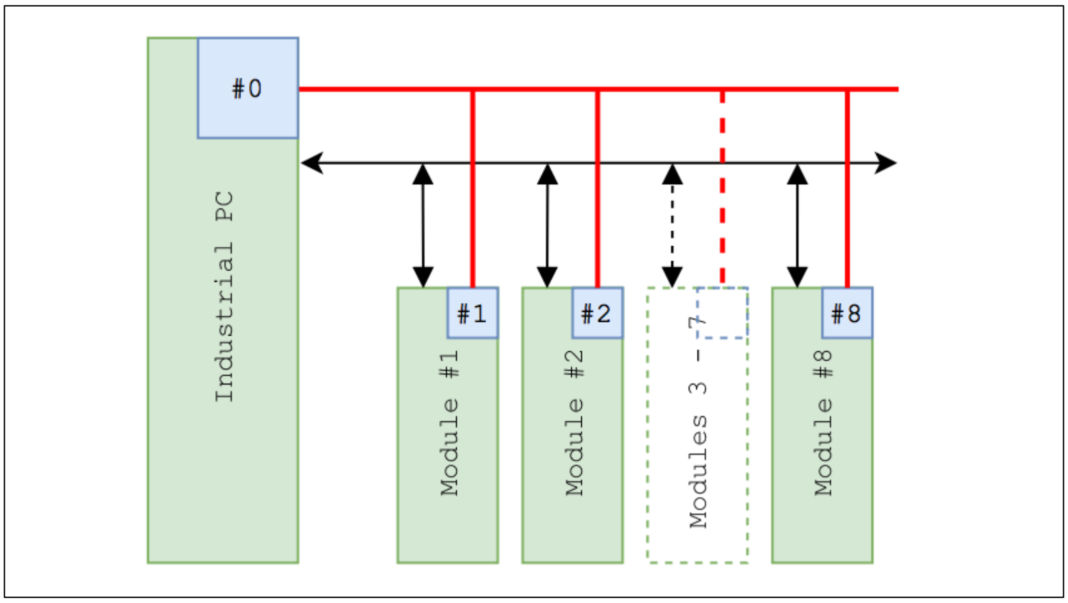


Figure 11. Simplified MCS block diagram (green – module/industrial PC, blue – current measurement block, red – power supply bus, black – CAN bus).

A simple fuse would be the most obvious choice, but while it is a very functional current protection mechanism, the non-resettable nature of it becomes cumbersome in applications where overcurrent can occur without a permanent device failure. In such situations, a fuse can be used in conjunction with a resettable current protection mechanism. Once the resettable current protection triggers, it must remain in a triggered state until a reset occurs. This reset can be implemented as hardware switch/button or a power-cycle (device is powered down and up again).

Over-current protection circuit should keep functioning whenever MCS module is supplied with power. Protection circuit is wired between module power supply input and protected area where MI module circuitry resides. See Figure 12.

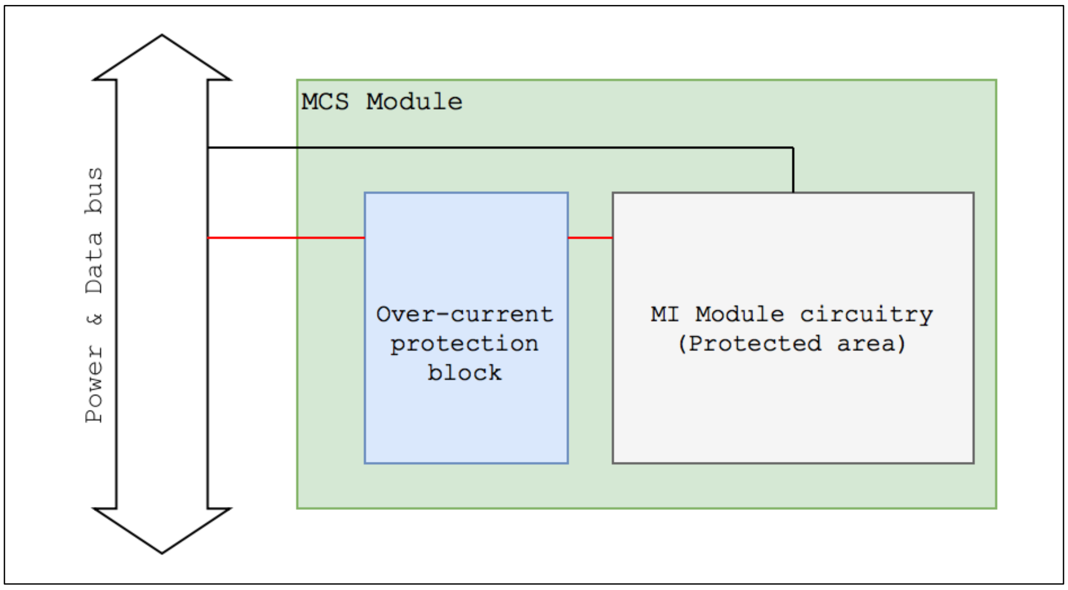

Power & Data bus
MCS Module
Over-current protection block
MI Module circuitry (Protected area)

Figure 12. Simplified MCS current protection diagram (black wire – CAN bus, red wire – power bus, blue – over-current protection block, gray – protected area).

Current protection schematic consisting of four operational amplifiers, MOSFET switch, sense resistor, and several other passive components is proposed as shown in Figure 13. For full schematic refer to Appendix 1. When an over-current condition is detected by primary comparator, it sets the latch comparator, which in turn de-energizes Q1. Even when the current flow has stopped due to open circuit, the latch comparator keeps Q1 de-energized.

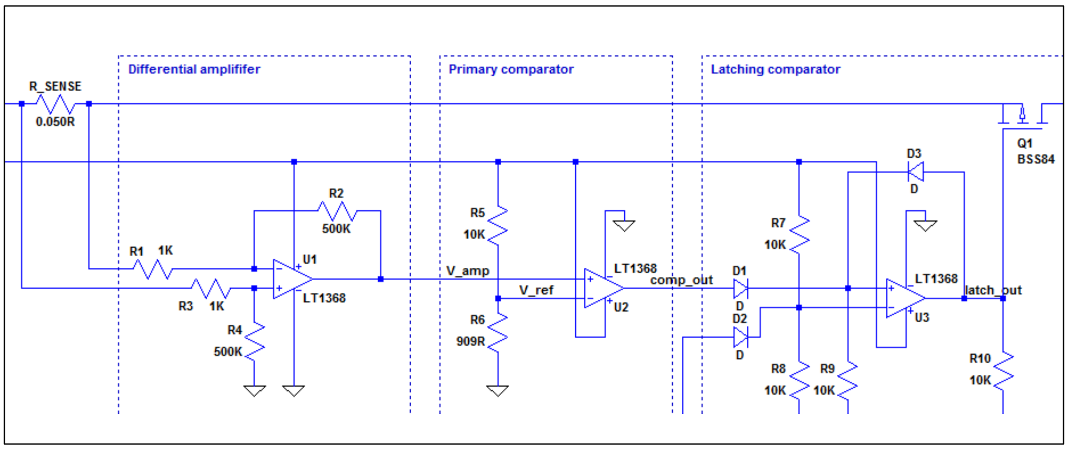


Figure 13. Main portion of over-current protection diagram.

Current flowing through sense resistor R_{SENSE} causes voltage potential difference across it. This voltage difference is amplified by a differential amplifier. Output voltage of amplifier is given by Equation 1.

Differential amplifier output calculation [8]

$$\mathrm{V_{AMP}} = \mathrm{V_2} * \frac{\mathrm{R4}}{\mathrm{R4+R3}} * \frac{\mathrm{R1+R2}}{\mathrm{R1}} - \mathrm{V_1}\frac{\mathrm{R2}}{\mathrm{R1}}. \tag{1}$$

Equation 1 can be simplified for well-chosen biasing resistor values. Consider,

$$\mathrm{R2} = \mathrm{R4},$$
$$\mathrm{R1} = \mathrm{R3},$$
$$\frac{\mathrm{R2}}{\mathrm{R1}} = \frac{\mathrm{R4}}{\mathrm{R3}}.$$

Thus,

$$\mathrm{V_{AMP}} = (\mathrm{V_2} - \mathrm{V_1}) * \left(\frac{\mathrm{R2}}{\mathrm{R1}}\right) = (\mathrm{V_2} - \mathrm{V_1}) * A, \qquad \text{where } \mathrm{A} = \frac{\mathrm{R2}}{\mathrm{R1}}. \tag{2}$$

Differential amplifier’s output is connected to primary comparator’s non-inverting input. This comparator is used to detect when an over-current condition has occurred. Comparator’s reference voltage is set by a voltage divider consisting of R5 and R6, as shown in Equation 3.

$$V_{REF} = \frac{\mathrm{R6}}{\mathrm{R6+R5}} * V_{in} \tag{3}$$

Primary comparator’s output is further fed into the secondary comparator a.k.a. latch comparator. It is used as a latch to ensure, that once an over-current protection is triggered, it remains in an active state, even after the high side switch has been de-energized and current flow has decreased below triggering level.

Latching action of secondary comparator is achieved using a positive feedback loop. Once the voltage on non-inverting input becomes higher than on the inverting input, output is driven to supply voltage V_{IN}. This causes diode D3 to conduct and keep non-inverting input at a higher voltage than the inverting, regardless of output from primary comparator. Latch comparator output level can be reset to GND level by removing supply voltage V_{IN} or by driving the inverting input's voltage level above the non-inverting.

The output from the latch comparator is wired to the gate pin of PMOS switch Q1. When comparators output is close to GND, the Q1's gate to source voltage V_{GS} is negative and above V_{GS} threshold, therefore, Q1 is conducting. However, when comparators output is close to V_{IN}, the Q1's V_{GS} voltage is close to 0 volts and below V_{GS} threshold, therefore, Q1 is not conducting.

The relationship between V_{REF} and the corresponding current flowing through R_{SENSE} can be calculating by applying Ohm's law

$$(\mathrm{V}_2 - \mathrm{V}_1) = I_{\mathrm{SENSE}} * R_{\mathrm{SENSE}} \tag{4}$$

To Equation 2, thus

$$V_{AMP} = I_{\mathrm{SENSE}} * R_{\mathrm{SENSE}} * A. \tag{5}$$

Maximum V_{AMP} voltage before over-current protection triggers is V_{REF}. Therefore,

$$V_{AMP(MAX)} = V_{REF} \tag{6}$$

Thus,

$$V_{REF} = I_{\mathrm{MAX}} * R_{\mathrm{SENSE}} * A. \tag{7}$$

The nominal current for modules is roughly 65 mA. The following values are chosen, and calculations done to set the output current I_{MAX} = 80 mA.

$$\mathrm{A} = 500, \qquad \text{and}$$
$$\mathrm{R1} = \mathrm{R3} = 1\mathrm{K}, \qquad \text{therefore}$$
$$\mathrm{R2} = \mathrm{R4} = \mathrm{R1} * \mathrm{A} = 1\mathrm{K} * 500 = 500\mathrm{K}.$$

Primary comparator's reference voltage V_{REF} according to Equations 6 and 7 is,

$$R_{\mathrm{SENSE}} = 0.050\ \mathrm{Ohm}, \qquad \text{and}$$
$$I_{\mathrm{MAX}} = 0.080\ \mathrm{A}, \qquad \text{therefore}$$
$$V_{REF} = 0.080 * 0.050 * 500 = 2\ \mathrm{V}.$$

Using Equation 3 the resistor R5 and R6 values are chosen as such,

$$R5 = 10\mathrm{K}, \qquad \text{therefore}$$
$$R6 = \mathrm{R5} * \frac{1}{\left(\frac{V_{in}}{V_{ref}} - 1\right)} = 10\mathrm{K} * \frac{1}{\left(\frac{24}{2} - 1\right)} = 909\ Ohm.$$

To ensure that the over-current protection is in non-triggered state upon module power-up, a Power on Reset (POW) circuit is used. See Figure 14. The POR circuit keeps the inverting input of U3 at logic one state for a short period of time when module is first powered up. This causes the latch comparator to be at a reset state.

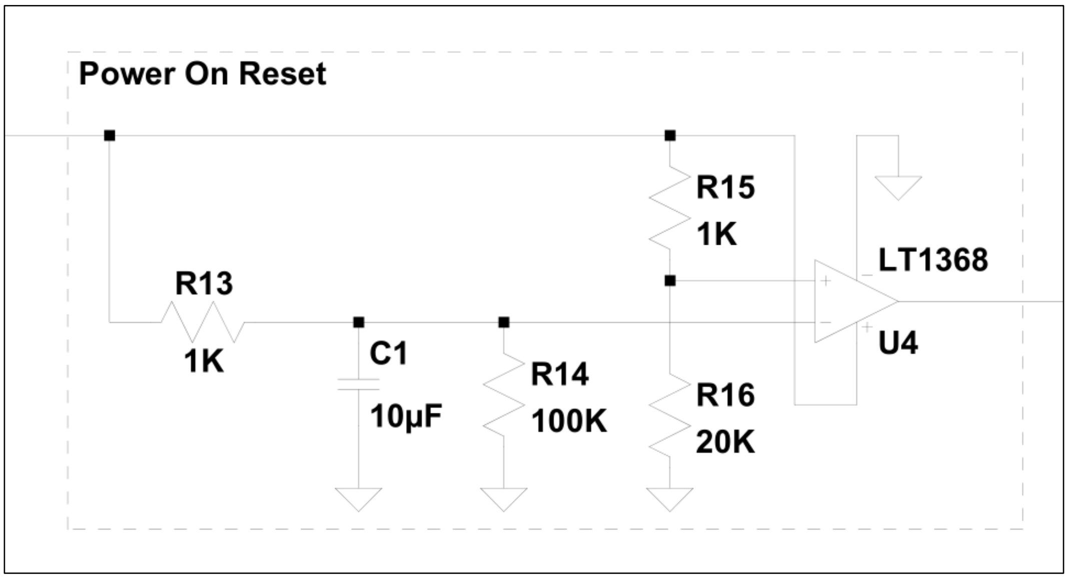


Figure 14. Power on Reset circuit.

It is constructed using a resistor-capacitor (RC) circuit and a comparator. Upon power-up capacitor C1 is charged until it becomes saturated. Comparator consisting of U4, R15, and R16 is used to convert charging capacitor potential into logical values. Initially operational amplifier’s output is at logic one. Once the inverting input of U4 becomes higher, than the non-inverting input, output becomes logic zero.

This type of implementation uses only four operational amplifiers, and a few discrete semiconductors and passive components, therefore, it can be implemented in a small physical footprint, and with a relatively high confidence of reliability.

Over-current protection circuit is simulated in LT Spice IV. See simulation graphs in Appendix 2. Simulation is five seconds long with power supplied to module after one second, and over-current fault condition introduced after three seconds from start of the simulation. As shown in graphs, over-current protection does indeed remove power from module and keeps it unpowered even after the fault condition has disappeared.

3.2 Controller Area Network (CAN)

3.2.1 Existing Application Layer Protocol

Multichannel User Interface uses a proprietary Application Layer protocol called MTR CAN. Full protocol specification is Janesko confidential, therefore, only a brief overview and the relevant portions of this protocol specification are given in this chapter.

MTR CAN defines behaviour of standard data frames being sent over CAN bus. It is responsible for formatting of data fields, creation of message IDs, and setting the message acceptance filter within nodes.

Specification defines that every node on the bus requires a node identifier (node ID), which is used to identify which node is the recipient of a CAN message. Node ID of the recipient node is referred to as target identifier (target ID) whereas node ID of the message sender is called sender identifier (sender ID). All nodes on the network have their message acceptance filters set so that only message frames whose target ID matches the node ID are accepted.

Message identifier (11-bit number residing in the arbitration field of a standard frame) is a concatenation of target ID, Object ID, and a read/write bit. See Figure 15.

Message ID										
Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Target ID					Object number					R/W bit

Figure 15. Concatenation of message identifier.

Data is packed into message frame data field according to predefined formatting instructions called Objects. Each Object has a defined Object ID. If a node receives a message with an unknown Object ID, then the message is ignored.

The read/write bit signals node whether the unpacked data should be written to its memory, or should an answer message be sent containing data packed according to Object specified by Object ID.

An example Object is shown in Figure 16. Byte seven of all Objects contains the same information – sender ID and an answer bit (identifies whether the message is a response to a request).

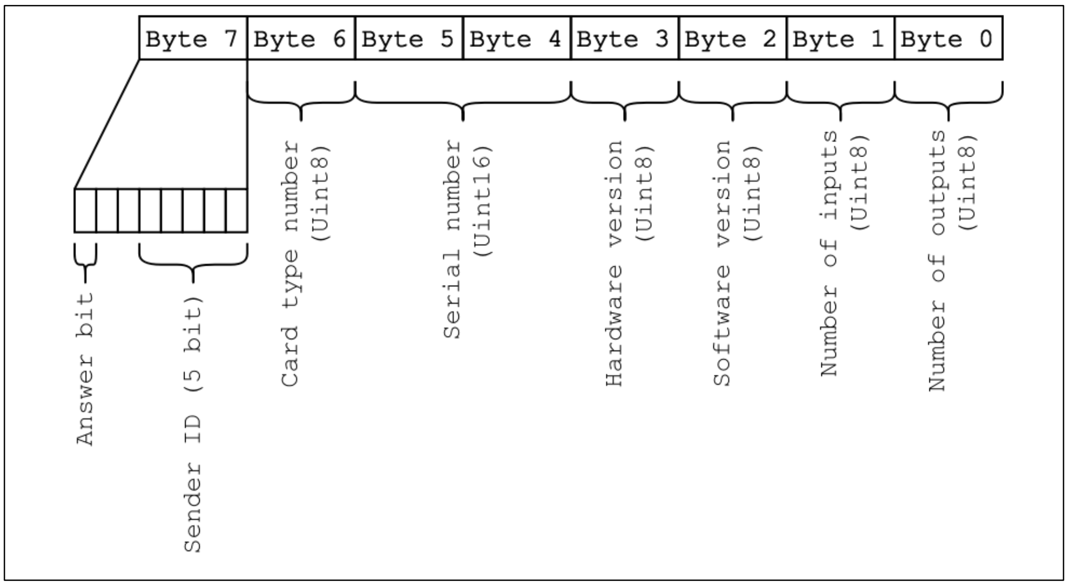


Figure 16. Example Object definition (Status Object).

Due to the node and Object ID's width being limited five bits, only maximum of $2^5 = 32$ nodes and Objects are supported by this iteration of MTR CAN specification.

3.2.2 Extended Application Layer Protocol

In order to implement a dynamic node ID distribution scheme as covered in chapter 3.2.3 Dynamic node identifier distribution, additional number of message identifiers is required. This is because the 11-bit message identifier used in MTR CAN has all the bits already defined. Fortunately, the extended 29-bit message format of CAN specification can be used to increase the total number of unique message identifiers available.

Additionally, to address the limited number of nodes and Objects possible according to MTR CAN specification, an extended MTR CAN Specification is proposed, which would use the extended CAN message frame format. This would increase the total number of possible CAN message identifiers from $2^{11} = 2048$ (11-bit identifier), to $2^{29} = 536870912$ (29-bit identifier).

Many CAN transceivers limit the maximum number of nodes on the bus to about hundred for 1 Mbps transfer speed. For this reason, the width of node ID (and subsequently target and sender IDs) is chosen to be seven bits. This gives maximum node number of $2^7 = 128$.

Message ID for extended frame is concatenated as shown in Figure 17.

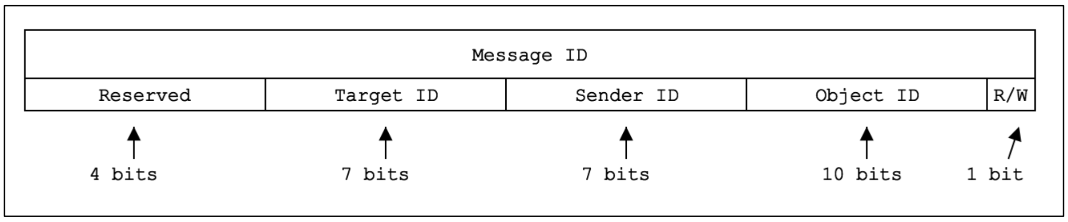


Figure 17. Concatenation of message identifier for extended frame.

The four most significant bits of message identifier are reserved to indicate special functionality messages. They are referred to as Special Purpose Bits (SPB) hereafter. When these bits form a value of seven (0x07), normal functionality is defined. That is, target, sender, and Object IDs are extracted from message ID and corresponding actions are taken.

3.2.3 Dynamic node identifier distribution

Often it is not possible to distribute node IDs during installation of the system. An example of such case would be when nodes can be added onto the CAN network dynamically, which is the case with both MI and MCS. The Multichannel User Interface distributes node IDs using the physical location of module.

In practice this is achieved by having three signals wired to each module through slot connector, which gives module a three-bit binary number to read. This number is unique for each slot as it is hardware set using pull-up and pull-down resistors on the motherboard. Module generates its node ID upon power-up by reading this hardware set connector slot number and applying a hardcoded offset.

However, this approach is not usable on MCS due to the lack of motherboard. The physical location of node cannot be known, therefore, a dynamic node ID acquisition scheme is devised to ensure that each node on the bus has a unique node ID.

Current design of MI and proposed MCS always has a single de-facto master node on the CAN network, which houses the main processing unit of the whole system (industrial PC). This simplifies the node ID distribution scheme significantly as information about current network state can be stored onto the master node, and it can be used to distribute a node ID to each slave node (module).

Two particular data message types are defined: node ID request message (NIDREQ), and node ID response message (NIDRESP). NIDREQ is a data message with zero data bytes in data field. Message ID is used both for message collision avoidance and data transmission. It is formatted as shown in Figure 18. It is used by slave nodes to request a node ID from master node. For both NIDREQ and NIDRESP messages the SPB number must be equal to six (0x06).

Before slave nodes have acquired a valid node ID, the message acceptance filter is set so that only NIDRESP message is accepted.

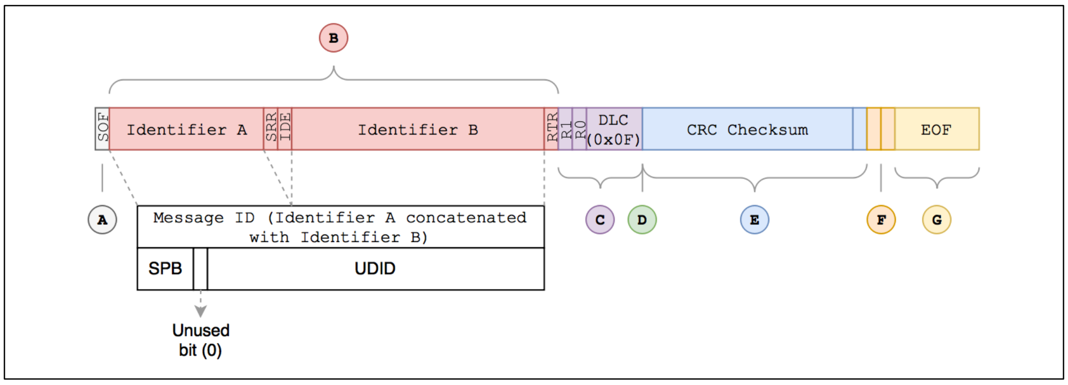


Figure 18. Node ID request message (NIDREQ) (A. Start of Frame, B. arbitration field, C. control field, D. data field, E. CRC field, F. ACK field, G. End of Frame).

NIDRESP message is used to respond to NIDREQ message. Its message ID is the same as the message ID of NIDREQ message it is responding to. The difference, however, is that the NIDRESP message contains two bytes of data in its data field. These bytes form a single unsigned 16-bit integer representing a node ID assigned to slave node. NIDRESP message is formatted as shown in Figure 19.

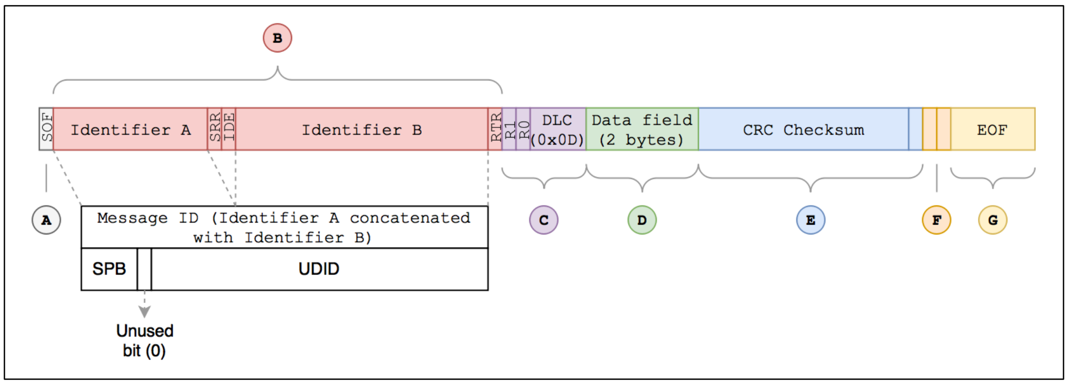


Figure 19. Node ID response message (NIDRESP) (A. Start of Frame, B. arbitration field, C. control field, D. data field, E. CRC field, F. ACK field, G. End of Frame).

Unique Device Identifier (UDID) is a unique identifier hardcoded into every device designed by Janesko Oy. This identifier is produced as a concatenation of 8-bit number defining device type and device 16-bit serial number. Therefore, UDID is a 24-bit wide number. See Figure 20.

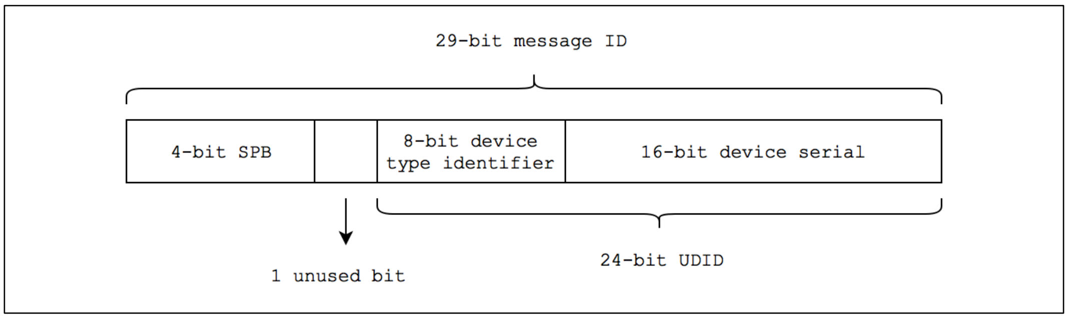


Figure 20. Message ID with alternate contents.

By placing UDID on the CAN message frame arbitration field it is possible to ensure that there are no conflicts taking place even when two nodes send a node request message at the exact same time. Bit arbitration will ensure that the device with lower UDID will win arbitration and continue message transmission.

Block diagram of the node ID distribution scheme is shown in Figure 21.

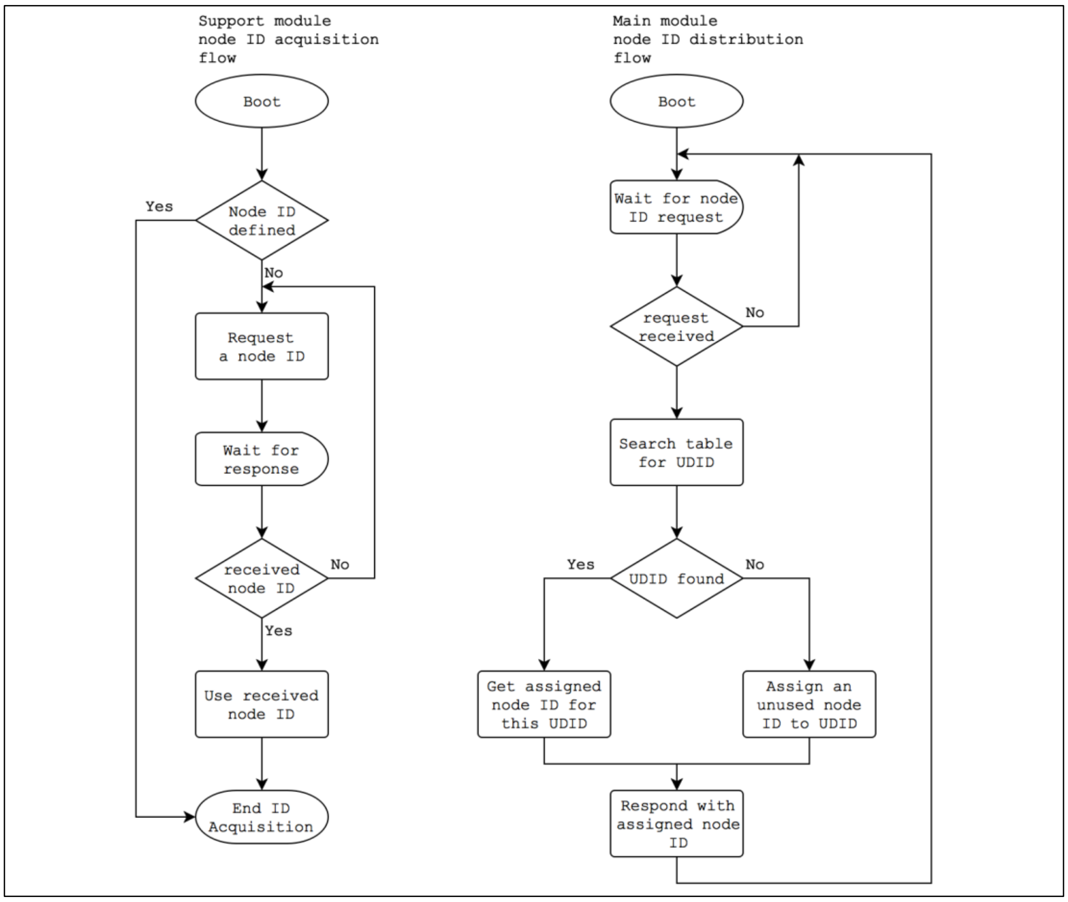


Figure 21. Flowcharts of Support module node ID acquisition (Right) and Main module node ID distribution schemes (Left).

Upon power-up slave nodes broadcast node ID request message (NIDREQ) and wait for response from master node. If no response is received within hardcoded timeout, then the NIDREQ message is repeated. This is done continuously until slave node receives a response from master node.

Master node keeps an ID table with UDIDs of slave nodes on the network, and corresponding assigned node IDs. When a NIDREQ message is received, master node extracts the slave nodes UDID from message identifier, and searches the ID table for it. If this particular UDID is found and already has a node ID assigned, then master transmits NIDRESP with the already assigned node ID. However, if the received UDID is not found, then a new entry is added to the table with this UDID and a newly chosen unique node ID assigned to it.

Code snippets from dynamic node ID acquisition are in Appendix 3. Because dynamic node ID acquisition was implemented into already existing code base, only the relevant functions and definitions are listed.

4 Conclusions

The goal of this project was to design a modular control system, which could be used with refractometers. An existing design of Multichannel User Interface was chosen as a basis. Two necessary modifications were identified and designed.

Firstly, a current protection circuit was designed to replace an existing current protection solution, which was not appropriate for this application. This circuit was simulated and confirmed to be functioning. Due to time constraints, no further testing and research was done for current protection. Possible further research that could be done is a comparison in functionality, physical dimension size, and price with the existing integrated circuit (IC) solutions.

Secondly, an addition to MTR CAN application layer protocol was developed to fulfil the needs of modularity (dynamic node ID distribution). Additional features may be added in the future to support more than one master module and increase robustness of dynamic node ID distribution.

Overall these design changes to existing Multichannel User Interface electronics and software allow for a creation of a new product which would fulfil the modularity needs end-users request. It is highly likely that within near future the MTR CAN extension either in its current form, or developed further by Janesko Oy, will be implemented into a commercial product.

References

1 K-Patents. Multichannel User Interface Instruction Manual. K-Patents; 2017. URL: http://www.kpatents.com/assets/files/downloads/manuals/pr-43/PR-43-MI-manual.pdf [Accessed on 10-04-2017]

2 K-Patents. K-Patents Multi-Channel User Interface MI, how to add a module video. K-Patents; 2016. URL: https://www.youtube.com/watch?v=LT1C4CqJVb0 [Accessed on 10-04-2017]

3 CAN in Automation (CiA). History of CAN technology. URL: https://www.can-cia.org/can-knowledge/can/can-history [Accessed on 10-04-2017]

4 Robert Bosch GmbH. CAN Specification, version 2.0. Robert Bosch GmbH; 1991. URL: esd.cs.ucr.edu/webres/can20.pdf [Accessed on 10-04-2017]

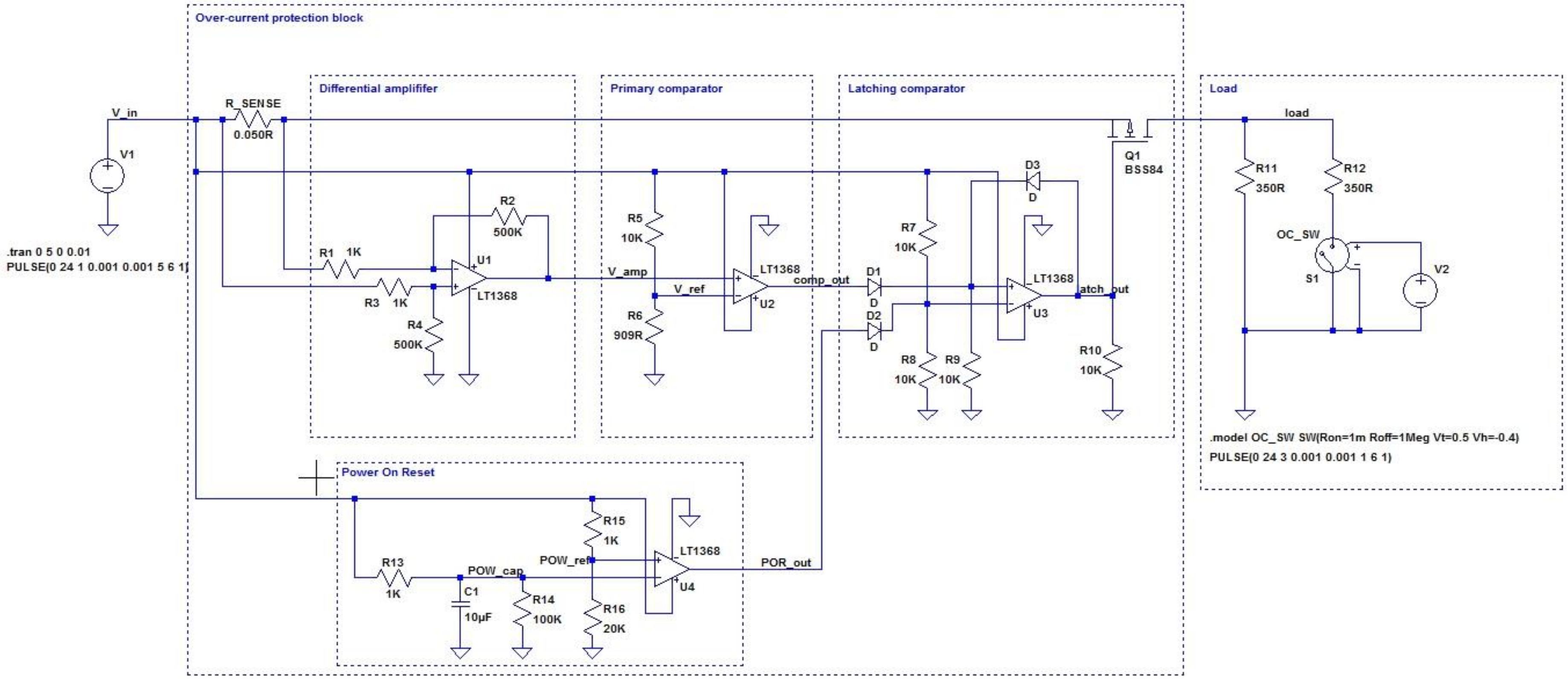
5 CAN Bus. URL: https://en.wikipedia.org/wiki/CAN_bus

6 Texas Instruments. Introduction to the Controller Area Network (CAN). Texas Instruments; 2002. URL: http://www.ti.com/lit/an/sloa101a/sloa101a.pdf [Accessed on 10-04-2017]

7 Network topology. URL: https://en.wikipedia.org/wiki/Network_topology

8 Differential amplifier. URL: https://en.wikipedia.org/wiki/Differential_amplifier

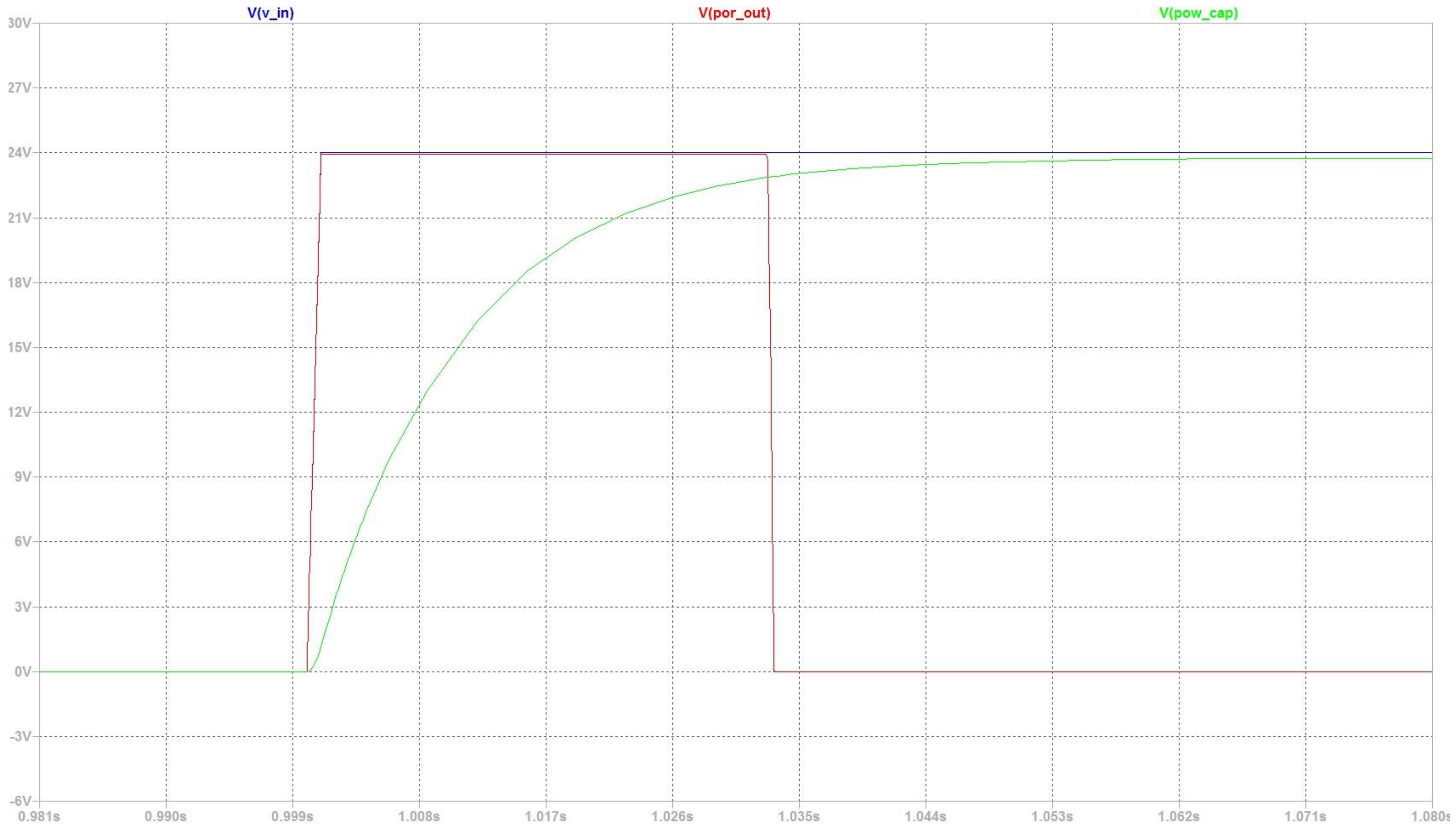
Over-current protection circuit simulation schematic (LT Spice IV)

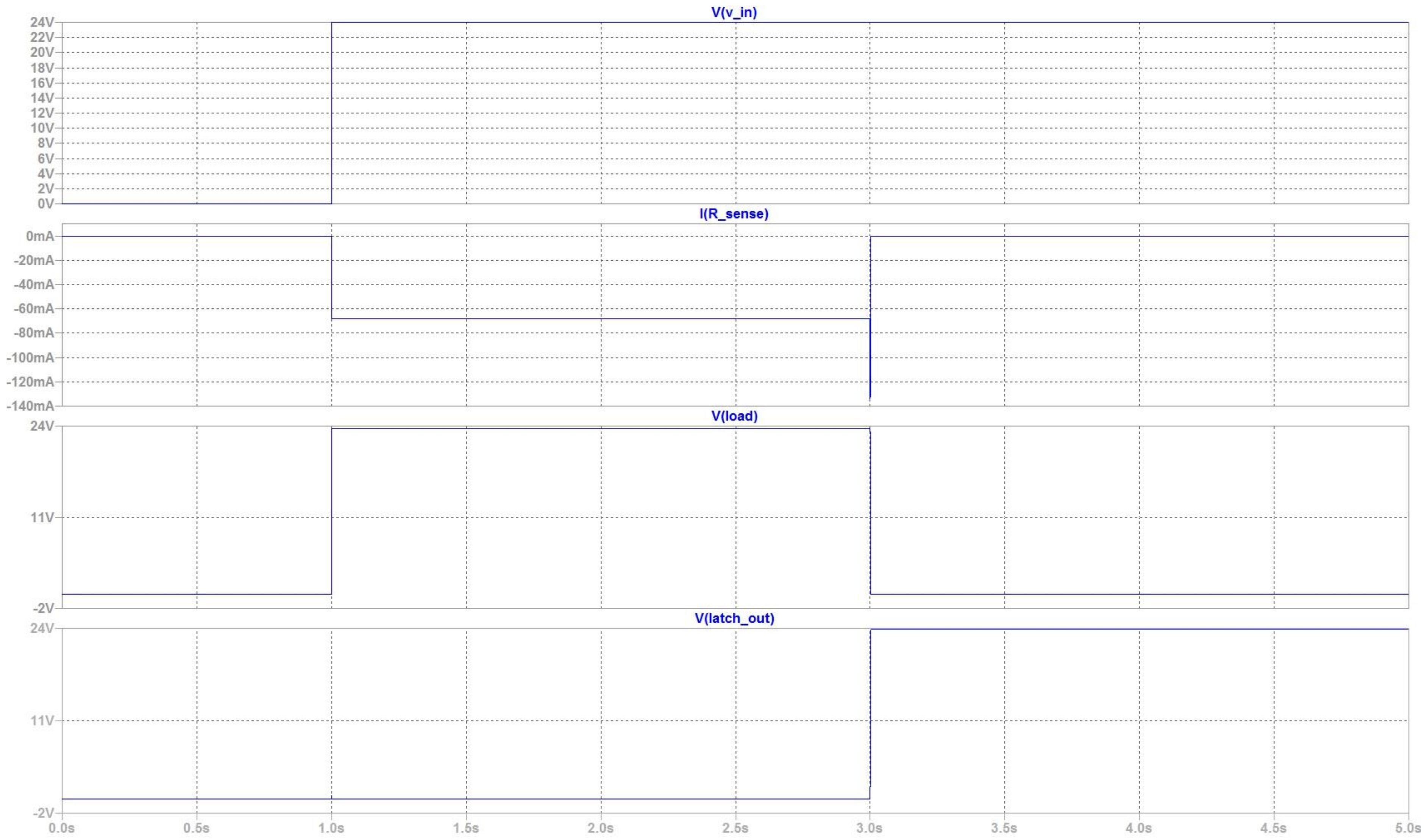


```
.tran 0 5 0 0.01
PULSE(0 24 1 0.001 0.001 5 6 1)
```

```
.model OC_SW SW(Ron=1m Roff=1Meg Vt=0.5 Vh=-0.4)
PULSE(0 24 3 0.001 0.001 1 6 1)
```

Over-current protection circuit simulation graphs (LT Spice IV)




V(v_in)
24V
22V
20V
18V
16V
14V
12V
10V
8V
6V
4V
2V
0V
I(R_sense)
0mA
-20mA
-40mA
-60mA
-80mA
-100mA
-120mA
-140mA
V(load)
24V
11V
-2V
V(latch_out)
24V
11V
-2V
0.0s
0.5s
1.0s
1.5s
2.0s
2.5s
3.0s
3.5s
4.0s
4.5s
5.0s

Dynamic node ID distribution code

```
uint32_t make_eid( struct canmsg *canr )
            {
            uint32_t new_sender_id = MAKE_SID( GET_TID( canr->id
));
            uint32_t new_target_id = MAKE_TID( GET_SID( canr->id
));
            uint32_t object = MASK_OBJ( canr->id );
            return( new_target_id | new_sender_id | object |
            SP_NORM);
            }

/*
 * Creates a unique identifier from card type and serial
 */
uint32_t make_udid(void)
            {
            uint32_t serial;
            uint8_t card_type;
            card_type = get_type();
            if( fram_read(( SN ), &serial, sizeof(serial) ) !=
sizeof(serial) )
                        debug( DEBUG_DEVELOPMENT_CAN, "FRAM read
failure - serial\n");

            return ((card_type << 16) | serial);
            }


/*
 * Acquires a node ID from master Node.
 */
uint8_t acquire_node(void)
            {
            struct canmsg canw;
            struct canmsg canr;
            static uint8_t node_id;
            if (node_id == 0)
                        {
                        canw.id =  SP_ACQ | make_udid();
                        canw.can2 = true;
                        canw.len = 0;
                        canr.id = canw.id;
                        canr.can2 = true;
                        canr.len = 1;
                        do
                                    {
```

```
                                    can_send( &canw );
                                    debug( DEBUG_DEVELOPMENT_CAN,
"\nSent ID request message\n");
                                    }
                        while(can_rcv( &canr, IDMASK_ACQ, 100 ));
                        debug( DEBUG_DEVELOPMENT_CAN, "\nNode ID
received\n");
                        node_id = canr.msg[7];
                        return node_id;
                        }
            else
                        {
                        return node_id;
                        }
            }

/*
 * CAN handling thread
 */
void can_thread(void)
            {
            struct canmsg canr;                 /* Struct for re-
ceived frame */
            waitsem(&startsem);                 /* Wait for UART
Thread to wake-up */
            delay(100);                         /* delay a second
to allow other modules to initialize */
            cprintf( "\n\nCAN Thread running (extended frames on-
ly)" );
            canr.id = get_node() | SP_NORM;
            canr.can2 = true;

            cprintf( "\nUsing ID: %u [0x%x]\n", GET_TID(canr.id),
GET_TID(canr.id) );
            for(;;)
                        {
                        can_rcv( &canr, IDMASK_EXT, -1 );

                        cprintf("\nFrame received\n");
                        output_frame( &canr, DEBUG_DEVELOPMENT_CAN
);
                        cprintf("\n-> " );
                        toggle_can_led();
                        /* execute wanted object handler */
                        obj_handler( &canr );
                        }
                }
```

Listing 1. Functions “make_udid”, “acquire_node”, and “can_thread” from source file “can_objpar.c”.

```
uint32_t get_node(void)
            {
            return (acquire_node() << 18);
            }
```

Listing 2. Function “get_node” from source file “main.c”.

```
/*
 * Extended frame definitions
 */
#define GET_SPCL(id)     ((id >> 25) & 0x0F)
#define GET_TID(id)      ((id >> 18) & 0x7F)
#define GET_SID(id)      ((id >> 11) & 0x7F)
#define GET_OBJ(id)      ((id >> 1) & 0x03FF)
#define GET_RWB(id)      (id & 0x01)
#define GET_ANS(frame)   ((frame->msg[7] & 0x80) >> 7)
#define MAKE_TID(id)     ((id & 0x7F) << 18)
#define MAKE_SID(id)     ((id & 0x7F) << 11)
#define MASK_OBJ(id)     (id & (0x03FF << 1))
#define IDMASK_EXT       ((0x7F << 18) | (1 << 29) | (15 << 25))
#define SP_NORM                                (8 << 25)

/*
 * Extended frame definitions for dynamic node ID acquisition
 */
#define SP_ACQ           (7 << 25)
#define IDMASK_ACQ       (SP_ACQ | 0xFFFFFF)                 /*
MASK used for node ID acquisition*/
```

Listing 3. Constant and Macro definitions from header file “can_objpar.h”.