# HomeAutomation project documentation

Thiemo Nordenholz

HA-D02.01
development
2016-08-23

# Notes

This documentation is currently under construction.

If contradicting protocol specifications are found on the project web page, this document is authoritative.

If contradicting implementation in software or hardware is found in the real world, it should be carefully considered whether the concepts or documentation is erroneous, or the implementation needs to be changed.

Sections of the documentation marked as ”Status: Production” should reflect their real-world implementations, and are considered correct. Here, a decision to change the real-world implementation (instead of the documentation) should be considered especially carefully.

Numbers prefixed `0x` are hexadecimal, `0b` designates binary, and no prefix indicates decimal numbers.



This document is typeset in DejaVu using LaTeX.

# Contents

# Part I.

# Concepts and definitions

# Concepts and definitions

Documenting the HomeAutomation project components requires a documentation of the underlying concepts as well. To present a well-defined vocabulary for the following descriptions, some technical definition of terms is also necessary.

In the following section, certain details are omitted for easier understanding - for example, events can not only be caused by hardware, as will be written here, but also by programs like the Timer Event Generator, a Remote Procedure Call (RPC) interface, or as a result of macro execution. Such details will then be presented in detail in the respective section, for this example, in `ha_control`'s own chapter.

# 1. Control Software concepts

The Control Software is responsible for the logical connections between events that have been detected by hardware and actions that have actual influence on the physical world.

Single devices (end devices, e.g. light switches or output channels) are addressed by identifiers in the format `[S.]RR.NN`, where `RR` is a room number, and NN a consecutive number of the device. Switches carry the `S.` prefix, output channels are not prefixed. This format is called the “room.number address”.

On the CAN bus infrastructure, they are identified by the CAN device address and a channel number on that CAN device. Translation between the two levels of identificators is provided by components of the control software. The concept of “channels” is explained below.

I/O devices can be grouped together for (pseudo-)simultaneous addressing by higher-level software. The interpretation of group definitions and the enqueueing of the devices contained therein is done by `ha_control`.

Input devices are linked to output actions - the easiest example being a light switch directly linked to the respective output control channel. Input devices can also control groups of output devices. They are also managed by `ha_control`.

Input activity or other event sources (like sensor readings, time of day, or even a pseudo-random number generator) can also trigger complex actions, which are defined as macros. These are written in a simple macro language, allowing output actions to be executed in several ways. Macros, again, are managed and executed by `ha_control`, which may fork subprocesses for the actual execution of the macro commands.

Continuous monitoring of the said sensors, by the way, is not needed as these are to report to the controller when they find something of interest. How exactly this is implemented is depending on the type of sensor in question, but the definition of actions on certain sensor reportings is also a matter of `ha_control`. How it does all this, and how its architecture looks like, is documented in the respective section of this document.

## 1.1. Bus devices

The HA system is organized along two CAN bus lines over which informational and command datagrams are transmitted. Actual control of, for example, lamps,

electrical outlets, or other household appliances, is executed by microcontroller-based circuits attached to one of the CAN bus lines. Each such bus participants (but not the HA control station) is called a “bus device” in this document.

## 1.2. HomeAutomation Control System, HACS

While the bus devices can operate independently from a central control instance because they can all listen to information on the bus that has been sent from another bus device, and can then operate according to their configuration, there is a PC-based system running software to coordinate the responses to certain events on the bus, to configure the devices according to user input, and to provide methods for the user to interact with the HomeAutomation software and hardware stack. This PC-based system is called the HomeAutomation Control system (HACS). It will be described in detail later in the specific component sections.

## 1.3. Actors

Bus devices do not directly interact with household line voltage levels, but command state changes to third-party devices which have undergone official certification processes for inclusion in such electrical circuits. These third-party devices that actually switch line voltage levels as commanded by bus devices are called “actors”.

Other switching devices not directly attached to the CAN bus that directly control other aspects of HA appliances are also called “actors”, regardless of whether they are self-developed or third-party.

## 1.4. Events

Events are changes to either physical conditions or logical state that are detected by HomeAutomation devices and made known to other parts of the infrastructure to that actions can be initiated in response.

Physical conditions, for example, could be a light switch being toggled, a sensor measuring a change in room temperature, or a window being opened. Examples for logical events that have no direct physical origin are a certain time of day at which an event is configured to be triggered, a web page having been accessed, or a phone call coming in.

Physical events are typically transported through the HA-CAN bus infrastructure, logical events are generated by the software stack within the HA control system. Direct insertion of events into the HA control software from other computers or

in general over an IP network connection is not supported due to security considerations. Therefore, all interfaces to the HA system that shall be available through the deployment site internal network or the Internet have to be decoupled from those networks by means described in the respective software sections.

## 1.5. Actions

Actions are the counterpart to events: An action changes either the state of some physical device connected to the HA system, or causes a change in logical state.

Examples for physical actions are a the throwing of a relay to turn on a lamp, opening shades on a window, or switching on a garden pump and opening a valve, watering the lawn. Logical actions include setting state variables within the HA software stack, sending a short message to the owner's mobile phone, or changing an icon's colour on a web page.

Actions are initiated by the HA control station by sending action commands to either software components under its control, or by sending CAN datagrams over the HA-CAN bus. In that case, bus devices detect the commands addressed to them, and act upon them, leading to the desired changing of state. The `ha_control` program consumes events and triggers actions according to its configuration, which is described in detail in the respective section starting on page 72.

Actions can also be initiated by CAN actor devices on their own, when Direct Event Actions are supported by them, have been appropriately configured by the control software, and the respective trigger event has been detected on the bus.

## 1.6. Channels and I/O ports

When needed in a context requiring bus communication, bus devices, as mentioned before, are identified by their CAN device address and a channel number. In many cases, such as for switching an output device on and off, the channel number is identical with the number of the I/O port on the device controller board.

However, certain actors require more than one output ports to control the attached appliance. For example, in the way the HA infrastructure manages them, shades control requires one switch to turn the motor on and off, and one to switch its direction. For such cases, the concepts of "channels" and "I/O ports" are defined:

A "channel" is a device number on one bus device under which the control software addresses the device when sending bus datagrams, and in that way, acts as an abstraction layer for the one or more actual I/O lines that a controlled appliance may need to be controlled. Thus, a switchable wall outlet consists of one channel, which the bus device firmware maps to one I/O port; a shades motor also

is addressed as one channel, but the bus device firmware controls two I/O ports according to commands it receives for this channel number.

XXX *XXX*This does not match the actual implementation.

Note that this requires that all the actual knowledge about how to control an attached device is delegated to the bus device firmware: If a device has to control attached shades motors, it has to know both how to sequence the toggling of on/off and up/down commands, and on which of its I/O ports the two functions are actually connected. This principle not only applies to shades control, but to all types of activities that a bus device is commanded to control for any type of attached appliance.

The first of these requirements, knowing how to control a given appliance, has to be implemented in the bus device firmware before deploying it to control such appliances.

The second requirement, knowing about the I/O port attachment of a specific device type, is configurable in the control station's device configuration, and can be updated by device management commands sent to the bus devices.

## 1.7. CAN bus

The "CAN bus" (or "bus") adheres to the published CAN specification, details of which are not documented here but in the respective official standards. A higher layer protocol specifically developed for the HomeAutomation project is defined in this document. The physical layer, consisting of cabling, connectors and voltage levels, is also defined here, in the respective sections.

As will be seen in the higher level protocol description, there is a bus number encoded in bus device addresses used in the control software. CAN itself specifies only the addressing scheme for one bus line, so this bus number is stripped of the addresses before transmission occurs on the CAN bus. However, the device interfacing the CAN bus to the HA control system is designed to manage two separate CAN bus lines, each with the full range of device addresses available on the bus, and each operating independently from the other.

This design was chosen to increase the security of certain bus devices. If one bus line is implemented so that it leaves the controlled building (for example, to communicate with outside or detached bus devices), this might allow an unauthorized intruder to tap into the CAN bus and listen to bus traffic, gaining information not intended for foreign knowledge, or even worse, send their own CAN commands and remotely control devices within the building.

A typical example scenario is the CAN-attached doorbell panel being unmounted by an attacker, who thereby gains access to the bus wiring, and sends a command to the access control system, electrically unlocking and opening the door - without involvement of any access checks implemented in the control station. This is mitigated by having the doorbell panel attached to one, "insecure", bus line, and

the door lock system attached to the second, “secure”, bus, which is never routed outside the walls of the controlled building.

For the purpose of sending CAN messages to it, a bus device is identified by its bus number and a device identifier (DID). However, the bus number is removed from this address when the message reaches the Dual-CAN/USB interface after it has selected the correct bus controller. The actual CAN message identifier is then calculated from the DID and a message type identifier (MTID). This whole process of address calculations is defined and described in detail in the Higher Layer CAN Protocol, HA-I02, on page 89.

# 2. Remote access

Remote access to the HomeAutomation control system has to be limited due to security considerations. The risk of unauthorized external parties remotely controlling household appliances has to be minimalized. Because security issues are assumed to exist in the self-written software as well as in the underlying operating system code, and recent (as of 2014) publications make it clear that even hardware is not neccesarily trustworthy when it comes to remote access backdoors, the HomeAutomation control systems are planned to be as isolated from the rest of the world, especially the Internet, as possible.

However, a certain, well-defined method of communicating with the HA system and the devices it controls is desired for a group of authorized users. This enables control of actions within the home and monitoring of system, device and environmental status from remote locations, as for example on vacation or in the workplace.

## 2.1. Functionality

The following functionality is provided by the HA remote access software, ordered by descending priority:

### 2.1.1. Remote reading of sensor data

Analog and digital sensors attached to the HA system can be read, either as textual raw data, or through a graphical user interface (GUI).

### 2.1.2. Remote control of actors

Actors can be remote-controlled by either a command-line interface, or a GUI. Not neccessarily all actors need to be accessible by the remote control mechanism - for example, certain functionality that might impair safety or security of the building or the users should be excluded from remote control.

### 2.1.3. Health monitoring of actors, bus devices, and HA infrastructure

The operational status of actors, CAN bus devices and its internal architecture, and the cabling and power levels can be monitored, either through a textual interface, or the GUI. Certain preconfigured corrective actions can also be initiated by the remote access mechanisms, however, unlimited permission for changes to the infrastructure is not allowed. For example, powering off a CAN bus is prohibited, while a fixed procedure to power-cycle it might be supported while limitations on the rate of such power-cycles to be initiated remotely are enforced by the control software. Firmware upgrades of bus devices are generally prohibited to outside channels.

### 2.1.4. Control station monitoring

The HomeAutomation control station's operaktional status and detected possible error situations can also be monitored through the outside access methods mentioned.

## 2.2. Security considerations

This section describes the security requirements defined for the remote control functionality. "Outside" in this section means any device which is not the HA control station (HACS) itself (or any CAN attached bus device). This might be the Internet, a telephone data connection, or any other means of electronically connecting to the HA system.

### 2.2.1. Direction of data flow

The outside is not allowed to initiate a connection to the HACS on its own. The HACS may periodically query its known communication partners for requests or actions to be executed, but can also cease such activity at any time when, for example, its computing resources are too loaded with internal work.

The periodic queries are currently planned not to take place over an Ethernet or IP connection, but through a serial line between the web server attached to the Internet and the HACS. This keeps possibly existing vulnerabilities in the HACS operating system's network stack from affecting the HA system security.

### 2.2.2. Data formats

Data exchanged between the outside and HACS must adhere to strict protocol definitions. For all data to be transferred from or to outside systems, datagram format, field lengths, and delimiters, as well as the length of individual datagrams are fixed. By placing these restrictions on the protocols used, common weaknesses in the receiving side software like receive buffer overflows are avoided.

Reception of invalid datagrams can be an indication of attempted attacks against the HACS, and as such, must be reported, preferably out-of-band, to an operator so that investigation and, if necessary, countermeasures can be initiated.

# 3. An example: Switching on a room light

To get an overview of the different components in the HomeAutomation system and their functionality, this section will follow a user action through the HA stack with reasonable detail: A user presses an in-wall switch to turn on the light in a room.

This explanation will neccessarily include more technical details than have been discussed so far, for example message identifiers exchanged on the bus, and hardware component details. If required, these can be looked up in the alphabetical index, or found in the later chapters of this documentation.

A digital input controller frequently checks its connected inputs for state changes. As long as the user presses the pushbutton light switch, this causes one of the controller's input lines to read a high level, which is a change from the default low level. Therefore, the controller sends an `INDIGPULSE` message on the CAN bus, informing the HACS about that change in state.

This message is read from the bus by the CAN/USB interface attached to the HACS, and converted into a serial line message. This in turn is read by `ha_iod` on the HACS, and dispatched to its TCP clients.

One receiving TCP client is `ha_control`'s iodclt thread, which is now responsible for generating an event message from this datagram, and placing it on the queues of other `ha_control` threads that have registered for notification about such events. For compatibility with the user-centric configuration file, the CAN-bus identifier (bus number, DID and channel number) is converted to the room.number identifier when creating the event notice. As layer of abstraction on the input side, the `INDIGPULS` CAN notice is translated to a `switch_toggled` event.

At least the actor thread is then notified about this event.

The actor thread has a table of possible events and the associated actions as configured through the `actions.xml` configuration and associated configuration files, in particular macros and group configurations. As a single light is to be switched, macros and groups do not apply for this example, though, so the action to cause is just the immediate switching of a single device. This device is identified by its device identificator (room.number type), and the matching Device instance (in this case, a `relayoutputclass` Device instance) is looked up.

The actor thread has been configured by the user to create an `output_toggle` action when a `switch_toggled` event is received for the respective light

switch device, so it calls the `Device.action_trigger()` method with the `action='output_toggle'` argument.

The `Device.action_trigger()` method creates from this an action command as a CAN datagram (actually, a list of CAN datagrams containing only a single datagram). The `Device` class knows its own address in both room.number and bus+DID+channel forms. The created datagram is addressed to the CAN address of the digital output controller to which the actor for the lamp is connected, the MTID is set as `OUTDIG`, and `data[0]` to the channel number. `data[1]` is set to 3, the `toggle` command. The completed datagram is then handed to the iodclt thread for transmission down the stream.

The iodclt thread sends the datagram through its TCP connection to `ha_iod`, which in turn reformats it and sends it through the USB-serial line to the CAN/USB controller. This device creates an actual CAN message from it, identifies the bus number to use, places it in the respective CAN controller's transmission buffer, and initiates the transmission. As soon as the bus is free to be used (or the HACS' CAN interface wins the bus arbitration process), the datagram will be transmitted to the actor bus device.

On the actor bus device, the message is received and evaluated, resulting in the identification of the I/O port for which a newly set state is desired. First, the current state of the port is measured through the actor's feedback line. Because the toggle command was received, this state is inverted to get the actually desired new state. The respective I/O port control signal is asserted to switch the actor, and the feedback line is monitored until the desired state has been entered, or a timeout has been reached. Then, the I/O port control signal is deactivated again, and a final measurement is done on the feedback line. If all components work, this measurement gives the desired new port state, and the respective room is illuminated now.

If the I/O port is found to be in the desired state, the bus device sends a `SYSM_LO_C, CI=S_IOSTATE` CAN message for the respective channel number and including the new state value to signal either `on` or `off`. If the final feedback line measurement did not confirm the desired state, a `SYSM_LO_C, CI=S_OFAIL` message is sent instead, and the I/O port is marked as failed in non-volatile memory in the bus device. If a timeout has occurred while driving the actor, but the final measurement confirms that the new state has been activated on the actor, a `SYSM_LO_C, CI=S_ODEGRAD` message is sent, and the I/O port is not registered as failed. In that case, operator interaction would still be expected to mitigate the imminent risk of the output device terminally failing.

The `SYSM_LO_C` message sent by the bus device is received by `ha_control` as detailed above, and handed to the Device instance which commanded the change in state by calling `Device.can_msg_handler()` with the respective event message as parameter. There, the internal state of the bus device is updated accordingly (Device.currentstate is set to reflect the new state of the actor).

If an error indicator (`SYSM_LO_C, CI=S_OFAIL` or `CI=S_ODEGRAD`) was received instead, appropriate error reporting would be initiated by `ha_control` on request

of the Device instance.

# Part II.

# Hardware

# 4. Infrastructure

The electrical infrastructure of this project regularily has to interact with AC line voltages. Such devices underly regulatory requirements, and may pose a threat to safety of the operator or other users, and therefore must be carefully designed.

To address both legal and technical requirements for devices attached to the public electricity network and to have the line voltage levels constructed and built as safe as possible, it was decided to only use Commercial Off The Shelf (COTS) parts where handling of potentially dangerous voltages is required. All self-designed parts of this project must be protected from contact with line level voltages by at least one level of COTS-provided isolation.

All self-designed parts of the project, on the other hand, deal with low voltage levels (Extra Low Voltage per IEC 60449) only. COTS Safety Extra Low Voltage (SELV) power supplies are used to generate the operating voltages for this project.

## 4.1. Extra Low Voltage Cabling

| *ELV cabling*: Production |
|---|

### 4.1.1. Actor CAN device to line voltage switch

Cabling that actuates line voltage switches needs to contain four signal lines: Two for the switching pulses, and two for the switch feedback.

Cable type $4 \times$ J-Y(St)Y, $2 \times 2 \times 0.8\,\mathrm{mm}^2$telephone cabling was selected for this purpose.

This cable is colour-coded yellow-red-black-white. The following assignment was made:

| | |
|---|---|
| yellow | +12V switch pulse |
| black | GND switch pulse |
| red | +12V feedback signal |
| white | GND feedback signal |

*Note:* The temporary direct cabling between light switches and switching devices that is installed during the contruction of the HomeAutomation infrastructure

to have a working installation uses different cabling. For this purpose, stranded wire 1 × 1.0mm²with red and violet coloured insulation is used. Here, red wire is used for the switch signal (+12 V DC), violet wire is the return path.

### 4.1.2. Sensor CAN device to installation light switch

Cabling that connects the digital pulse input CAN devices to the actual light switches in the building needs to contain three signal lines: One for each of the two included switches, and one signal common path.

Because of this type already having been specified for the actor CAN device to line voltage switches, the same cable type 4 × J-Y(St)Y, 2 × 2 × 0.8 mm²telephone cabling was selected for this purpose.

This cable is colour-coded yellow-red-black-white. The following assignment was made:

| | |
|---|---|
| yellow | right side switch signal |
| black | common |
| red | (unused) |
| white | left side switch signal |

*Note:* Some switches that have been installed early in the project implementation phase use different cabling and different colour assignments. These are documented in the floor plan (`EG_ELT_Schalter` layer), and not part of this project description.

### 4.1.3. CAN bus cabling

The CAN bus wiring is constructed using cable type 12 × J-Y(St)Y, 6 × 2 × 0.8 mm (solid wire) telephone cable. The cable currently in use consists of six pairs of twisted single wires, where five of the six pairs are made up from one coloured wire insulation together with a white wire. The remaining pair is coloured red and blue, following a German colouring code for telephone wiring.

The following assignment of colours to numbers and signals is made (numbers in brackes are the original telephone wire names):

| | | | | | |
|---|---|---|---|---|---|
| red | 1 (1a) | CANH | blue | 2 (1b) | CANL |
| white | 3 (2a) | GND | yellow | 4 (2b) | +12V |
| white | 5 (3a) | RSVD_5 | green | 6 (3b) | SHIELD |
| white | 7 (4a) | GND | brown | 8 (4b) | +12V |
| white | 9 (5a) | not used | black | 10 (5b) | not used |
| white | 11 (6a) | not used | blue | 12 (6b) | not used |

Wires marked “not used” may be connected to CAN bus devices which may or may not connect them internally on both CAN bus terminal blocks. In any case, it must be ensured that these wires are electrically connected on every bus line. This means that, if they are left unconnected from a bus device, they must be connected by means of splicing connectors, preferably spring clamps.

## 4.2. AC Line Voltage Switches

| *AC line voltage switches*: In use |
|---|

The switching of AC line voltage has been specified to be initiated by pulses of 12 V DC. As the line circuits are protected by 16 A fuses, a current rating of at least 16 A is required for the switches.

A secondary requirement was the inclusion of an isolated parallel switch in the actor to provide feedback for the controlling electronics: The switching impulse needs to be as long as required for the switching action, but should not be unneccesarily long.

Furthermore, it was desired that the switching actuators consume power only while they are changing their state, thus latching relays were specified.

Another secondary requirement was a method for manual throwing of a switch in case of HA infrastructure failure, or when the HA systems are turned off for maintenance.

For on/off switching of line voltage, *Hager EPN521* devices fulfil these requirements.

For double throw switches, *Hager EPN528* devices were selected.

On both switching devices, the left hand side channels are used for line voltage, the right hand side channels are used as feedback contacts, switching a 12 V DC control voltage.

Cable colour assignment is described in section ”actor CAN device to line voltage switch” on page 21.

# 5. Bus devices

Development of bus devices usually is done in well-defined steps. First, a schematic drawing is designed, and a concept for the device firmware is either documented or implemented. When the schematic has reached a state which can fulfil at least a basic functionality, it is build on a prototype material, usually “breadboard”. From the results of this development setup, the schematic is refined further, and in parallel, the firmware to run on the system is debugged and improved. When a state of usability is reached that allows for attachment of the device to the HA-CAN bus, and the hardware design allows for all intended purposes of the device, a prototype is build on a production-ready printed circuit board (PCB), which is either manufactured in-house, or ordered from a PCB manufacturer. This prototype is used for further refining of the firmware and bus interaction of the device. If insufficiencies in the hardware design are found, they are reworked into the prototype PCB, and the schematic drawing is changed to match. Afterwards, depending on the amount and impact of changes, a new prototype PCB may be produced to be used in the further development. If hardware, firmware, and the interaction with the then-current revision of other bus devices is found to be sufficient, the required number of production PCBs is built, again either in-house or by order from a commercial board manufacturer.

This section documents those bus devices that have reached at least the prototype state.

The HA-B part number registry contains several part designators that were abandoned (at least for the time being), but still allocate a part number. These registrations remain in place to avoid collisions in file system and revision control system names.

# 6. HA-B02, Dual CAN/USB interface

| |
|---|
| *HA-B02.02*: Development<br>*HA-B02.01*: Production |

The dual CAN/USB interface attaches two HomeAutomation CAN bus lines to the HomeAutomation Control System (HACS) through a Universal Serial Bus (USB) connection. The interface acts as a USB “device”, so the “bus master” capabilites of the HACS are required.

## 6.1. HA-B02.02

### 6.1.1. Compatibility

Firmware HA-P04.01 and higher is compatible with this hardware revision.

Support for CAN bus power measuring is available in firmware HA-P04.02 and higher.

### 6.1.2. Changes from previous revision

1. Change CAN bus connectors to include two `+12V` pins according to the updated standard
2. Add current sensing circuitry for CAN bus power lines
3. Change power connector from 6-pin to 4-pin header

### 6.1.3. Description

#### Power supply

Power is supplied to the device from an external power supply through connector `J3`. Continuous current requirements are, by specification, 2 A on the 12 V rail, and 1 A on the 5 V rail.

Note that the 12 V rail, which powers the devices attached to the CAN bus lines, is protected by a 1 A fuse on each bus, but when choosing a power supply for

HA-B02, some margin should be included. A current handling capability of for example 5 A on the 12 V rail together with overcurrent protection in the power supply should be enough.

*+12 V supply rail*

The +12 V supply is exclusively used to provide 12 V power to CAN bus devices through the bus connectors. Because there will be interference and coupling into the 12 V cabling on the bus lines, power conditioning must be done on each bus device individually anyways, and no further filtering has been included on HA-B02.

*+5 V supply rail*

The +5 V rail powers all logic devices on the board. It is buffered by one board-wide 4.7 µF electrolytic capacitor, with additional ceramic capacitors located at the individual ICs as required.

Note that no further protection against overload or voltage spikes is included. This should be included in the power supply itself, which, due to the importance of this component for the overall HomeAutomation system, might be a battery-backed system anyways, and therefore already provide current limiting and over-voltage protection.

*Microcontroller unit*

The ATmega32 microcontroller is operated at +5 V supply voltage. Local buffering capacitors of 100 nF are present on all supply inputs. The analog-to-digital converter (ADC) supply is connected through a 10 µH inductor as well, as recommended by the Atmel application note AVR042, “AVR Hardware Design Considerations”.

The analog reference voltage output `MCU_AREF` is also decoupled through a 100 nF capacitor (`C14`) to `GND`.

*CAN bus power supply*

CAN bus devices are provided with +12 V, 1 A total operating power through each of the two bus lines. This supply is taken directly from the +12 V input connection, but can be switched on and off individually per bus by software. A 1 A fuse is included in the supply lines for each CAN bus to interrupt the power supply in case of short circuits, mainly due to miswirings or erroneous handling.

Power switching is done by a latching relay, so power consumption is limited to the actual state change. The specified relays can turn off 1 A of current, and were the limiting factor on bus line current specification. However, together with the good availability of the chosen bus wiring, this limitation matches the wiring’s

current capablities well. Maximum switching voltage is 110 V, and is therefore well above the system's requirements.

### 6.1.4. Part specifications

The following parts have been specifically selected for this design. Other parts used are considered uncritical.

| Part | Description | Manufacturer: model |
|---|---|---|
| IC1, IC4 | CAN transceiver | Microchip Technology: MCP2551 |
| IC2, IC5 | CAN controller | Microchip Technology: MCP2515-I/P |
| † | CAN controller | Microchip Technology: MCP2510-I/P |
| IC6, IC7 | 20V/V current shunt amplifier | Analog Devices: AD8211-2 |
| IC100 | USB-UART interface | FTDI: FT232RL |
| R33, R35 | Current shunt resistor | *Any*: 0.1 R, 0.5 %, 0.5 W four-terminal |

† Table entries marked with a "dagger" symbol list alternative parts that may be used without requiring changes to hardware or software.

### 6.1.5. MCU

While low latency and reliable message throughput is important for the device's functionality, the firmware operation is not too complex, because dedicated ICs take care of the actual protocol handling on both the CAN and USB side.

An Atmel ATmega32 MCU has been selected for the purpose, It is, at the time of design, one of the larger ATmega type MCUs, containing 32 kB of flash program memory and 2 kB internal SRAM. CLock frequencies up to 16 MHz are possible. This design uses 30 of the 32 available Input/Output lines.

*Clock generation*

For easy clock generation on the UART interface, the MCU clock was specified at 11.0592 MHz, which is generated by the external crystal `Q4` on the ATmega's `XTAL` pins. Load capacitances of 22 pF are added against `GND`.

*Reset*

The $\overline{\texttt{MCU\_RESET}}$ signal is the MCU's primary reset source. Under normal conditions, it is pulled high by the 10 k `R22` so reset is disabled. Through the MCU programming header `JP3`, it can be pulled low by an external programmer, which is required to enter flash memory programming mode.

The MCU's power-on reset function is programmed to delay processor startup for *XXX* check on hardware. XXX

The on-chip brownout detection (BOD) circuit is configured to trigger a BOD re-
XXX set if the supply voltage falls below *XXX* check on hardware.

### 6.1.6. CAN bus interfaces

The two CAN bus interfaces are designed to be identical, each having their own controller, transceiver, connector, and power management circuitry.

The bus interface is controlled by the MCU through an MCP2515 controller IC, which handles all the CAN protocol conformance, and provides a simple programming interface to the firmware.

The MCU can select one of the MCP2515 controllers by enabling the respective slave select signal, SPI_CS_CANA on pin PB4, or SPI_CS_CANB on PB1.

*Clock generation*

Both CAN bus controllers are supplied with a 16 MHz clock, generated by the crystal oscillator module QG1, which is capable of driving two clock inputs without further driver circuitry.

*Reset*

The MCP2515 reset signals, CANx_RESET, are pulled low (active) through R16 (CANA) and R29 (CANB). They can be controlled by the MCU on PB0 and PB3, respectively.

These signals need to be driven high by the device firmware in order to activate the CAN controller.

*Bus transceiver*

The physical bus interfaces are implemented in the MCP2551 CAN bus transceiver IC, IC1/IC4.

Their operation can be controlled from the CANx_TCVR_STBY signal connected to the general purpose I/O pin 10 on their respective CAN bus controller IC.

If not driven, these signals are pulled low through 100 k resistors and acts as a slope-limiting control input. Please refer to the MCP2551 datasheet for details on this.

MCU firmware can instruct the MCP2515 CAN controller to pull CAN_TCV_STBY high to place the transceiver IC into a listening-only “sleep mode”. For normal operations, the CAN controller’s GPIO pin 10 should be kept in a high-impedance state.

*Bus current measurement*

The +12 V power supply to each CAN bus connector is routed through a 0.1 R current shunt resistor to enable measurement of the individual bus current consumption. The voltage drop over this resistor is fed into a 20x measurement amplifier, and the resulting voltage is made available on microcontroller inputs to the analog-to-digital converter (ADC).

As no external voltage reference is provided, it is suggested to run the ADC from the internal 2.56 V reference. This bandgap-referenced voltage is supposed to be more stable than the supply voltage, which would be the other alternative reference.

With this configuration, the actual current on one bus line can be calculated from the (16-bit) ADC output value as $I_{\text{bus}} = 1.25 n_{\text{ADC}}$ mA.

### 6.1.7. USB interface

To interface with the HACS, the device uses an FT232 UART-to-USB interface IC (IC100). It provides 5 V UART levels to the MCU, and is connected to the respective UART input, which transmits data on PD1 and receives data from USB on PD0.

*Clock generation*

Generation of all required clock signals is internal to the FT232 interface IC.

*Reset*

The $\overline{\text{USB\_RST}}$ signal is connected to a voltage divider consisting of R7 (4.7 k) and R12 (10 k). This is fed by the USB connector's voltage input and referenced to GND, resulting in the USB controller being held in reset when no USB host is connected, and the reset signal being deasserted when a USB host is attached and providing power to the device.

$\overline{\text{USB\_RST}}$ is also available on the MCU's I/O pin on PA5, and can be used to find out whether the USB connection is currently present. Because the signal is connected to both the USB supply voltage and GND through resistors as described, the MCU can also drive the $\overline{\text{USB\_RST}}$ if the reset state needs to be controlled by firmware.

*Bus connector*

The USB bus connector is specified as standard USB-B receptacle. Other form factors may be specified without further modifications to the device or its firmware if required.

### 6.1.8. Interfaces

*CAN0, CAN1, CAN bus connectors*

CAN0 connects to CAN bus A, CAN1 connects to CAN bus B.

Please see the HA-I07.02 specification for external CAN bus connectors on page 115 for the description of this interface.

*J3, power connector*

This is a “Molex”-type, 4-pin, 2.54 mm pitch, connector, providing power to the device. If required, any other functionally identical wire-to-board connector can be used.

Pin assignments for J3:

| | |
|---|---|
| 1 | +5V |
| 2 | GND |
| 3 | +12V |
| 4 | GND |

*JP2, I²C connector*

This 5-pin, 2.54 mm pitch header provides access to the MCU’s I²C interface. It can be used for debugging purposes and to attach future expansion peripherals.

Pin assignments for JP2:

| | |
|---|---|
| 1 | IIC_SDA |
| 2 | GND |
| 3 | IIC_INT |
| 4 | IIC_SCL |
| 5 | VCC |

IIC_INT is pulled down through 10 k resistor R8 and connected to the MCU’s pin PB2, which can be used to trigger interrupt request INT1.

*JP3, MCU ISP*

This is a 6-pin, 2.54 mm pitch header conforming to Atmel’s standard pin assignment for 6-pin ATmega ISP headers.

Pin assignments for JP3:

| | |
|---|---|
| 1 | SPI_MISO |
| 2 | VCC |
| 3 | SPI_SCK |
| 4 | SPI_MOSI |
| 5 | $\overline{\text{MCU\_RESET}}$ |
| 6 | GND |

*X2, USB connector*

This is a standard USB-B receptacle.

Pin assignments for X2:

| | |
|---|---|
| 1 | USBVCC |
| 2 | D+ |
| 3 | D- |
| 4 | GND |
| 5 | USB shield (unconnected) |

*LEDs*

Four LEDs are present on the device, pairwise associated with one of the CAN buses.

The LEDs are driven by a BS170 MOSFET to minimize load on the MCU I/O pins. With the current capability of the MOSFETs, this leaves plenty of room for connecting other LEDs or signal peripherals instead of the LEDs, if required by system design.

Pull-down resistors of 10 k to GND are present on the nets controlling the LED driver MOSFETs to prevent false triggering while the MCU's output pins are in high-impedance state before having been initalized.

| CAN | colour | MCU pin |
|---|---|---|
| A | green | PC3 |
| A | yellow | PC2 |
| B | green | PC5 |
| B | yellow | PC4 |

### 6.1.9. Jumper configuration

*CAN bus termination*

Wire bridges can be soldered into `J1` and `J2` to enable 120 R bus termination resistors on CAN bus A and B, respectively. If HA-B02 is physically located at the end of a bus cable, it needs to be active; if there are other devices on both sides of HA-B02's bus connection, the respective termination enable jumper needs to be disconnected (not stuffed, removed, or cut open).

## 6.2. HA-B02.01

Obsolete

### 6.2.1. Firmware

Firmware HA-P04.01 and higher is compatible with this hardware revision.

# 7. HA-B06, Modular CAN-I/O module, bus logic board

| |
|---|
| *HA-B06.04*: Production<br>*HA-B06.03*: Prototype, obsolete<br>*HA-B06.02*: Prototype, discontinued<br>*HA-B06.01*: Development, discontinued |

The CAN-I/O bus devices are designed with a modular approach in mind: One bus logic board attaches to the HA-CAN bus, manages the bus communications and handles the logical decisions required for changing actor state or reading sensor state information. A separate controller or sensor module is then responsible for interaction with the external actors or sensors, and isolates the HA system from the electrical infrastructure outside of it. This forms an additional layer of safety, together with the commercially sourced devices that are exclusively used when interaction with AC line voltages is required. The form factor of the modules, connector placement, and interface between bus logic board and controller/sensor boards is standardized and defined in HA-I11, starting on page 127.

## 7.1. HA-B06.04

### 7.1.1. Compatibility

Firmware HA-P05.02 and higher is compatible with this hardware revision.

This revision is usable with the following bus modules:

- HA-B07.02 and higher (16-way output module)
- HA-B08.02 and higher (16-way input module)

### 7.1.2. Changes from previous revision

The B06.04 revision corrected some design and layout errors made in the B06.03 prototype version:

1. Changed +5V to +3V3 on ATxmega ISP header and $\overline{\text{RESET}}$ pin

2. Corrected `BS108` footprint in EDA part library, updated board accordingly
3. CAN bus connector updated from HA-I07.01 to HA-I07.02
4. Removed TP7 and TP8 from UART RX and TX
5. Added TP9 and TP10 for `SPI_MOSI_5V` and `SPI_SCK_5V`
6. Placed `CAN_TCV_STBY` under MCU control
7. EDA part library improvements

### 7.1.3. Description

#### Power supply

The operating power is supplied through the CAN bus attachment by means of two +12 VDC lines, providing up to 1 A, but sharing this power budget with all bus devices attached to this bus line.

*+5 V supply rail*

Most of the circuitry on this device operates at +5 V.

For efficiency, a modular switch mode DC/DC converter module, `DC1`, has been specified for the conversion from 12 V bus power to the 5 V power rail. Power filtering capacitors have been included as suggested in the manufacturer's datasheet.

The DC/DC converter module is capable of providing a current of 1 A, which is sufficient for this device's operation.

XXX At the calculated maximum current requirement of *XXX* mA, thermal stress is not an issue, and additional cooling is not required.

*Microcontroller unit*

The ATxmega128A1U microcontroller unit (MCU) operates of a 1.6-3.6 V power supply for clock speeds up to 12 MHz, a supply voltage of at lesat 2.7 V is required for the full operating speed of up to 32 MHz.

Because this design clocks the MCU at 16 MHz, a supply voltage of 3.3 V was chosen for it.

The MCU's 3.3 V operating voltage is generated by a linear voltage regulator IC, `IC1`, running off the +5V supply net. The choice of a linear regulator for this purpose allows it to be placed near the MCU on the PCB

This IC is capable of providing 250 mA. The MCU is specified to consume 20 mA when active and running at 32 MHz. Because the MCU's output pins are designed to not directly drive peripherals without further driver circuitry, the I/O current

consumption is generally low, and the current budget of this voltage regulator is not exceeded.

Again, power filtering has been implemented as per manufacturer's recommendations.

*I/O ports*

The I/O ports available on this board connect to the outside only through the module interconnect (JP1) and the communications ports. No direct connection to high-power peripherals is available. Therefore, the I/O lines are driven from the general power supply rails, with certain protective circuitry where required, as described in the respective sections of this documentation.

A reference of the voltage levels used on the available peripheral connectors is available in the interface descriptions on page 40.

### 7.1.4. Level conversion

The device includes parts being powered from both the 3.3 V and 5 V rail that need to communicate with each other. None of the 3.3 V inputs are tolerant to 5 V levels, and the switching thresholds of the 5 V devices do not provide the option of driving them from 3.3 V directly. Level conversion circuitry is provided for the different supply domains.

*5 V → 3.3 V*

Four signals need to be converted from 5 V to 3.3 V levels: $\overline{\texttt{CAN\_INT\_5V}}$, I2C_INT_5V, SPI_MISO_5V, and UART_RXD_5V. All these nets drive only CMOS inputs, so the current requirements on them is very low and is mainly caused by pull-down resistors.

Simple voltage dividers were therefore designed for these nets, with 3.3 V zener diodes to GND on the 3.3 V side to protect against voltage spikes exceeding the designed input voltage.

*3.3 V → 5 V*

The 74HCT244 IC provides 8 driver circuits that can operate from the 5 V rail and provide equivalent TTL-compatible output levels. Their minimum high-state input voltage is specified at 2.0 V under these operating conditions, the maximum low-state input voltage at 0.8 V. It is therefore possible to drive the inputs from 3.3 V devices and have them correctly be recognized and propagated to the 5 V domain.

The IC's output enable pin (active low) is tied to GND through R2 (560 R) so the drivers are always enabled. Unused input pins are tied to GND through 10 k resistors to ensure stable behaviour.

*Bi-directional level shifting*

The I²C bus lines can be driven from either side of the level shifting circuitry, and therefore need to be translating levels bi-directionally. An n-channel MOSFET was used for this conversion, as shown and explained in the Philips application note AN 97055, “Bi-directional level shifter for I²C-bus and other systems”. `Q3` and `Q4` make up these shifters for the two I²C nets in this device.

### 7.1.5. Part specifications

The following parts have been specifically selected for this design. Other parts used are considered uncritical.

| Part | Description | Manufacturer: model |
|---|---|---|
| DC1 | 5 V, 1 A DC/DC converter | Recom Power: R-785.0-1.0 |
| IC1 | 3.3 V, 250 mA regulator | Microchip Technology: MCP1700-3302E/TO |
| IC2 | ATxmega128AU1 MCU | Atmel: ATxmega128A1U-AU |
| † | ATxmega128A1 MCU | Atmel: ATxmega128A1-AU |
| IC3 | Octal driver | *Multiple*: 74HCT244 |
| IC4 | CAN controller | Microchip Technology: MCP2515-I/P |
| † | CAN controller | Microchip Technology: MCP2510-I/P |
| IC5 | 64 kBit EEPROM | Microchip Technology: 24LC65P |
| IC6 | CAN transceiver | Microchip Technology: MCP2551 |
| Q3, Q4 | n-MOSFET, 200 V, 250 mA | *Multiple*: BS108 |
| † | n-MOSFET, 100 V, 320 mA | Diodes, Inc: ZVNL110A‡ |

† Table entries marked with a “dagger” symbol list alternative parts that may be used without requiring changes to hardware or software.

‡ Other alternatives may be selected according to the criteria given in Philips AN97055.

### 7.1.6. MCU

An Atmel ATxmega128 microcontroller unit was chosen for this device. It contains 128 kB flash program memory and 8 kB internal SRAM, provides over 70 Input/Output pins, and contains several peripheral functions. Furthermore, with an advanced interrupt and event system (“advanced” at least for microcontrollers of this size), it eases the development of firmware with the required functionality and reliability. The controller can operate at clock speeds up to 32 MHz.

Cost of the device is also acceptable, the only disadvantage being the availability in fine-pitch surface-mount packages only. However, these are manually mountable as tests have shown.

*Clock generation*

The MCU starts up with internal clock generation at 2 MHz by default. At this clock frequency, the firmware's initialization routines are executed.

It is then the device firmware's responsibility to configure the MCU's internal PLL circuitry to double the 8 MHz available from Q1, resulting in the desired operating core frequency of 16 MHz.

*Reset*

The $\overline{\text{RESET}}$ signal is the MCU's primary reset source. Under normal conditions, it is pulled high by the 10 k R30 so reset is disabled. Through the MCU programming header JP4, it can be pulled low by an external programmer, which is required to enter flash memory programming mode.

The MCU's power-on reset function is programmed to delay processor startup for *XXX* check on hardware. XXX

The on-chip brownout detection (BOD) circuit is configured to trigger a BOD reset if the supply voltage falls below *XXX* check on hardware. XXX

### 7.1.7. EEPROM

For storage of configuration data, a 64 kbit EEPROM device is connected to the I²C bus on TWIC. Its bus address is configured by hardware to 0x52.

### 7.1.8. CAN bus interface

The CAN bus interface consists of two connectors, CAN1 and CAN2, which have onboard pass-through connections for +12V, GND, CANH, CANL, and CAN_RSVD_2.

The bus interface is controlled by the MCU through an MCP2515 controller IC, which handles all the CAN protocol conformance, and provides a simple programming interface to the firmware.

*Clock generation*

The CAN bus controller runs at 16 MHz which is generated from the Q2 crystal.

*Reset*

The MCP2515 reset signal, $\overline{\text{CAN\_RESET\_5V}}$, is pulled low (active) through R17. It can be controlled by the MCU as $\overline{\text{CAN\_RESET}}$ on PB0 through level-shifting in IC3.

This signal needs to be driven high by the device firmware in order to activate the CAN controller.

*Bus transceiver*

The physical bus interface is implemented in the MCP2551 CAN bus transceiver IC, IC6. Its operation can be controlled from the CAN_TCV_STBY signal connected to PA6 on the MCU through level-shifter IC3.

If not driven, it is pulled low through 68 k R25, and acts as a slope-limiting control input. Please refer to the MCP2551 datasheet for details on this.

MCU firmware can pull CAN_TCV_STBY high to place the transceiver IC into a listening-only "sleep mode". For normal operations, device firmware should place PA6 in a high-impedance state.

### 7.1.9. Interfaces

*CAN1, CAN2, CAN bus connectors*

Please see the HA-I07.02 specification for external CAN bus connectors on page 115 for the description of this interface.

*JP1, module interconnect*

Please see the HA-I11.01 specification document on page 127 for the description of this interface.

*JP2, I²C connector*

The I²C connector is included for low-level configuration of e.g. the CAN bus address on initial startup, when no ACAA protocol is available. Current plans, however, move this configuration functionality to the UART interface because of more available COTS devices to attach to it.

The device includes 4.7 k pull-up resistors on both the 3.3 V and 5 V sections of both I²C bus lines.

Furthermore, it can be used for debugging purposes, or to attach further expansion peripherals.

Pin assignments for JP2:

| | |
|---|---|
| 1 | GND |
| 2 | I2C_INT_5V |
| 3 | +5V |
| 4 | I2C_SDA_5V |
| 5 | I2C_SCL_5V |

I2C_INT_5V is internally pulled low through a 10 k resistor (R4).

The connection between pin 4 and 5 and the actual I²C signal nets is protected against short circuits by 100 R series resistors R21 and R22.

*JP3, UART connector*

The UART connector is attached to the MCU's PORTC UART interface through the level-shifting circuitry as described above. The voltage divider consisting of D4, R20, and R23 translates the UART_RXD_5V signal available on the interface, and IC3 translates the UART_TXD to the externally available 5 V level.

To protect the device circuitry which is driving the signal, the UART_TXD_5V pin is short-circuit protected by a 100 R series resistor (R24), and protected against voltage spikes by a 1N4733 zener diode (D6) to GND.

The incoming signal (UART_RXD_5V) is level-translated as explained above, and the protective zener diode included in that circuitry (D4) will absorb overvoltages on this pin, with the voltage divider's upper resistor R23 acting as a overcurrent protection.

In this way, damage is also prevented if a user connects RS-232 levels directly to the UART interface (which is not advised), and communication might even work.

Pin assignments for JP3:

| | |
|---|---|
| 1 | GND |
| 2 | UART_TXD_5V |
| 3 | +5V |
| 4 | UART_RXD_5V |

Note that the "TX"/"RX" designators are assigned from an MCU viewpoint: The device MCU is receiving data from the outside on the UART_RXD line, it sends data out on the UART_TXD line.

*JP4, MCU PDI*

This interface is used to program firmware into the MCU. It adheres to the manufacturer's specifications, so regular Atmel programmer devices or compatibles can be used.

This interface operates on 3.3 V levels, as it is directly connected to the respective MCU pins.

Pin assignments for JP4:

| | |
|---|---|
| 1 | PDI_DATA |
| 2 | +3V3 |
| 5 | $\overline{\text{RESET}}$ |
| 6 | GND |

*LEDs*

Four LEDs are present on the device, pairwise associated with one of the CAN bus connectors.

The LEDs are driven by a BS170 MOSFET to minimize load on the MCU I/O pins, and to be able to operate them from the 12 V supply rail. With the current capability of the MOSFETs, this leaves plenty of room for connecting other LEDs or signal peripherals instead of the LEDs, if required by system design.

Note that there are no external pull-down resistors on the nets controlling the LED drivers, so a short flickering on power-on may occur. This is considered uncritical by the designer. Firmware should enable a logic low level on the respective pins on device initialization.

| CAN | colour | MCU pin |
|---|---|---|
| 1 | green | PQ0 |
| 1 | yellow | PQ1 |
| 2 | green | PQ2 |
| 2 | yellow | PQ3 |

*Test points*

The following test points are provided:

| | |
|---|---|
| TP1 | I2C_INT |
| TP2 | I2C_INT_5V |
| TP3 | I2C_SCL |
| TP4 | I2C_SCL_5V |
| TP5 | I2C_SDA |
| TP6 | I2C_SDA_5V |
| TP9 | SPI_MOSI_5V |
| TP10 | SPI_SCK_5V |
| TP11 | GND |
| TP12 | GND |



## 7.2. HA-B06.03

Obsolete

### 7.2.1. Compatibility

Firmware HA-P05.02 and higher is compatible with this hardware revision.

This revision is usable with the following bus modules:

- HA-B07.02 and higher (16-way output module)
- HA-B08.02 and higher (16-way input module)

### 7.2.2. Changes from previous revision

1. MCU changed from ATtiny88 to Atxmega128A1U
2. Module interconnect changed from 10-pin to 40-pin connector
3. Standardized PCB layout of module interface connectors specified
4. Complete redesign of supportive circuitry

## 7.3. HA-B06.01 / HA-B06.02

Obsolete

These development and prototype revisions include an ATtiny88 MCU as main microcontroller, using shift registers on the peripheral modules to address the actual I/O channels to infrastructure devices. This method turned out to be prone to errors in channel addressing and easily failed to meet the timing requirements, the module architecture was redesigned to include a wider module interconnect, and direct access to the channels from the BLB. This redesign increased the required number of I/O ports on the microcontroller on the BLB, and therefore a design based on the ATxmega family of controllers was created. The change in controller architecture also brought more computing resources on the BLB. Here, the increase in working memory (RAM) was especially useful, as it allowed a more robust software design, while at the same time increasing BLB software flexibility.

### 7.3.1. Compatibility

Firmware HA-P05.01 is compatible with this hardware revision.

# 8. HA-B07, Modular CAN-I/O module, 16-way output module

| |
|---|
| *HA-B07.04*: Development<br>*HA-B07.03*: Production<br>*HA-B07.02*: Prototype, obsolete<br>*HA-B07.01*: Development, discontinued |

The 16-way output module is connected to the Modular CAN-I/O Bus Logic Board (BLB) and controls up to 16 relay output actors, each switching a household electricity circuit through the bi-stable magnetic switches described on page 23.

## 8.1. HA-B07.04

### 8.1.1. Compatibility

This revision is usable with bus logic board HA-B06.03 and higher.

The BLB interface conforms to HA-I11.01.

*XXX* Chaining of output modules to a BLB requires certain firmware revisions on BLB and local MCU that are not yet defined. XXX

### 8.1.2. Changes from previous revision

The B07.04 revision places the I²C I/O-expander's bus address under microcontroller configuration and makes the I/O power return path switchable by the MCU.

This makes it possible to attach more than one output module to a single Bus Logic Board.

1. Route IC2 I²C address lines to MCU
2. Add MOSFET *XXX* and optocoupler *XXX* XXX XXX

## 8.2. HA-B07.03

### 8.2.1. Description

#### Power supply

Power is supplied to the device's logic circuits through the module interface connector. To power the externally connected household voltage switches, there exists a separate power rail which is optically isolated from the bus logic.

*+5 V supply rail*

On-board logic is in general supplied from the `+5V` supply net available on the module interconnect. It is filtered on the output board by a 4.7µF electrolytic capacitor.

*+3.3 V supply rail*

Circuitry interfacing with the Bus Logic Board (BLB) is powered from the `+3V3` supply net, also provided by the module interconnect. It is as well filtered by a 4.7µF electrolytic capacitor.

*+12 V I/O power rail*

The actor switch solenoids are powered from a local 12 VDC power supply, which is designed to be isolated from the HomeAutomation controller hardware. The `IRLD024` output driver MOSFETs are capable of switching 2.5 A, the spring-loaded connector terminals can withstand 2 A continuous current. Because the selected coils in the `Hager EPN` switches consume only 0.8 A, however, this +12 V power rail is equipped with a 1 A "medium" glass fuse.

#### Module interface

Please see the HA-I11.01 specification document on page 127 for the description of this interface.

An Atmel ATtiny88 MCU is connected to the module interconnect I²C bus, and serves as responder to module identification requests from the BLB as described in the HA-I11 interface specification. The MCU's $\overline{\texttt{MCU\_RESET}}$ signal is pulled up to `+5V` through a resistor internal to the ATtiny88. However, by programming port `P6` on the I²C I/O extender `IC2`, the BLB MCU can control this signal remotely as desired.

### Isolation

The two power supply domains (“logic”, coming from the module interface, and “power”, being connected from a local power supply at the deployment location) are separated by optocouplers for all signals that need to cross the domain boundary.

The actual line of this isolation is marked with a thick white line on the circuit board top side. Copper traces on the board are designed to never leave their respective domain. The selected four-channel optical isolation devices (TLP291-4 optocouplers) provide a minimum isolation voltage of 2.5 kV rms.

### I/O ports

All 16 I/O ports are designed identically. For this description, the part identifiers of the first port (“channel 1”) are referenced.

*Driver*

The +3.3 V signal coming in from the module interconnect is used to toggle the gate of n-MOSFET T113. During power-on of the bus device, the Bus Logic Board’s respective output is in high impedance state, so to prevent false triggering, this net (OUT_1) is pulled down to GND through 10 k resistor R121. T113 completes the ground path for the transmitter LED in optocoupler OC1D from supply net +5V when triggered from OUT_1.

When thus turned on, the phototransistor in OC1D conducts VCCIO through a 120 R series resistor (used to limit the transistor’s collector current) into the actual power switching n-MOSFET’s gate (T3). When turned off, the transistor’s emitter is grounded through a 5.6  resistor (R21) to GNDIO to keep leakage currents from triggering T3.

When triggered, T3 completes the current path to GNDIO for the relay’s coil coming in from the actor connector, pin 2, DRV_GND. The coil is powered from VCCIO on that connector’s pin 1, DRV_+12V. Diode D1 protects T3 from reverse voltage from the collapsing magnetic field in the relay coil, in addition to the MOSFET’s internal reverse diode.

*Feedback*

The relay’s feedback contact is powered from VCCIO through relay connector pin 3. When the relay switch is closed, this voltage is seen on pin 4, powering the transmitter LED in OC2A through current-limiting resistor R30. Thus, the phototransistor conducts the +3V3 supply net to the module interconnect’s feedback line, OUTFB_1. If OC2A is not conducting, leakage currents are grounded through 100 k resistor R11 to GND.

The LED currents of all 16 outputs' feedback circuits would add up to over 200 mA drawn from the 12 V supply rail if all output relay switches were powered on. Besides driving that rail to the limit of the 1 A fuse that was specified as soon as one relay is fired, this current consumption is unnecessary. The feedback optocouplers (`OC2, OC5, OC7, OC9`) have their transmitters not directly connected to `GND`, but to `OUTFB_GNDIO_n` lines, with `n` ranging from 1 to 4. The following paragraphs explain their control circuitry.

The module interconnect contains an I²C bus, through which the BLB's MCU can control circuits on the module device. I²C I/O extnder `IC2` is connected to this bus and is used to control four `OUTFB_EN_n` signals. BLB firmware needs to initialize these as low-level outputs on startup.

Continuing with the description of channel 1, `OUTFB_EN_1` is connected to the gate of `T109`, and connected to `GND` through 10 k pull-down resistor `R116` to ensure a stable state during the uninitialized high-impedance state on startup or with disconnected BLB. Similar to the design described in the "Driver" section above, `T109` controls optocoupler `OC3A`, which provides isolation from the I/O power supply domain. `OC3A` controls n-MOSFET `T1`, which, when activated and conducting, provides the `GNDIO` connection for `OUTFB_GNDIO_1`. This way, the transmitter LEDs in feedback optocoupler `OC2` will only activate when the respective feedback group is enabled by the BLB MCU.

BLB firmware should implement a sequential round-robin checking of the feedback lines, turning on only the `OUTFB_EN_n` signal required for the respective group being checked at one time.

### 8.2.2. Part specifications

The following parts have been specifically selected for this design. Other parts used are considered uncritical.

| Part | Description | Manufacturer: model |
|---|---|---|
| F1 | Glass fuse, 5 × 20 mm, 1 A, medium-fast | COTS |
| OC1 .. OC9 | Optocoupler: CTR≥50%, V(f)=1.2 V, I(f)=50 mA, V(CEO)≥25 V, I(C)≥20 mA | Toshiba: TLP291-4† |
| T3 .. T10, T17 .. T24 | n-MOSFET: V(DS)≥40 V, R(DSon)≤0.14 R @ 5 V, V(GSth)≤2 V, I(D)≥2.5 A | Vishay: IRLD024†‡ |

†Different devices that fulfil the given specifications may be used, but should be tested to work before deployment in a production environment.

‡Different pinouts are acceptable for alternate part choices.

## 8.3. HA-B07.02

Obsolete

### 8.3.1. Compatibility

This revision is usable with bus logic board HA-B06.03 and higher.

The BLB interface conforms to HA-I11.01.

### 8.3.2. Changes from previous revision

The B07.02 revision changed the module interface to HA-I11.01.

## 8.4. HA-B07.01

Obsolete

This development revision was planned for a development stage of the module interconnect using a 10-pin header and serial shift register protocol. A schematic of this board revision exists in the revision control system, a prototype was constructed and tested, and in consequence the module interconnect was redesigned. No B07.01 PCBs were manufactured.

### 8.4.1. Compatibility

This revision is usable with bus logic board HA-B06.01 / HA-B06.02.

# 9. HA-B08, Modular CAN-I/O module, 16-way input module

| |
|---|
| *HA-B08.03*: Prototype<br>*HA-B08.02*: Prototype, obsolete<br>*HA-B08.01*: Development, discontinued |

The 16-way input module is connected to the Modular CAN-I/O Bus Logic Board (BLB) and reads up to 16 low-voltage momentary switch contacts, for example light switches deployed within the house. The BLB generates CAN datagrams in response to state changes on the switch inputs.

## 9.1. HA-B08.03

### 9.1.1. Compatibility

This revision is usable with bus logic board HA-B06.03 and higher.

The BLB interface conforms to HA-I11.01.

### 9.1.2. Changes from previous revision

This revision changes a schematic error: $\overline{\texttt{MCU\_RESET}}$ is now pulled up to `VCC` through `R6`, not pulled down to `GND`.

### 9.1.3. Description

*+5 V supply rail*

On-board logic is in general supplied from the `+5V` supply net available on the module interconnect. It is filtered on the output board by a 4.7µF electrolytic capacitor.

*+3.3 V supply rail*

Circuitry interfacing with the Bus Logic Board (BLB) is powered from the `+3V3` supply net, also provided by the module interconnect. It is as well filtered by a 4.7µF electrolytic capacitor.

*+12 V I/O power rail*

The switch contacts are connected to `VCCIO`, which is fed from a locally connected +12 V power supply. COTS models which provide overcurrent and overtemperature shutdown have been selected and deployed for this purpose, there is no further protection built into this device.

### Module interface

Please see the HA-I11.01 specification document on page 127 for the description of this interface.

An Atmel ATtiny88 MCU is connected to the module interconnect I²C bus, and serves as responder to module identification requests from the BLB as described in the HA-I11 interface specification. The MCU's $\overline{\texttt{MCU\_RESET}}$ signal is pulled up to `+5V` through 4.7 k resistor `R6`.

### Isolation

The two power supply domains ("logic", coming from the module interface, and "power", being connected from a local power supply at the deployment location) are separated by optocouplers for all signals that need to cross the domain boundary.

The actual line of this isolation is marked with a thick white line on the circuit board top side. Copper traces on the board are designed to never leave their respective domain. The selected four-channel optical isolation devices (`TLP504A-2` optocouplers) provide a minimum isolation voltage of 2.5 kV rms.

### I/O ports

All 16 I/O ports are designed identically. For this description, the part identifiers of port 0/1 ("IO 0") are referenced.

A pair of switches is attached to connector `IO0`, with a common supply voltage line from `VCCIO` on pin 2, and the return paths coming in on pin 1 (`SW_RIGHT`) and 4 (`SW_LEFT`). The sense circuitry is identical for both switch signals.

When the switch is closed, `VCCIO` can flow through current-limiting series resistor `R1` to activate the optocoupler's transmitter LED. This lets the bus voltage side of `OC1A` conduct `+3V3` to `SENSE_0`, which is made available on the module interconnect

for evaluation by the BLB MCU. When OC1A is inactive, 100 k resistor R8 grounds leakage currents through the optocoupler's phototransistor.

Note that, unlike the HA-B07 output module's feedback lines, no round-robin activation of the transmitter LEDs has been designed for this device, because the switches to be used are only momentarily conducting and thus limiting the maximum simultaneous current flow.

### 9.1.4. Part specifications

The following parts have been specifically selected for this design. Other parts used are considered uncritical.

| Part | Description | Manufacturer: model |
|---|---|---|
| OC1 .. OC4 | Optocoupler: CTR≥50%, 1.0 V≤V(f)≤1.3 V, 16 mA≤I(f)*≤20 mA, V(CEO)≥25 V, I(C)≥1 mA | Toshiba: TLP504A-2† |

* Recommended operating condition

†Different devices that fulfil the given specifications may be used, but should be tested to work before deployment in a production environment.

## 9.2. HA-B08.02

Obsolete

### 9.2.1. Compatibility

This revision is usable with bus logic board HA-B06.03 and higher.

The BLB interface conforms to HA-I11.01.

### 9.2.2. Changes from previous revision

The B08.02 revision changed the module interface to HA-I11.01.

### 9.2.3. Engineering Change Orders

*ECO-016-004*

Released: 2015-06-30.

$\overline{\text{MCU\_RESET}}$ is erroneously pulled down through R6, thus keeping the MCU in permanent reset.

1. Keep un-stuffed or remove from the stuffed board the 4.7 k resistor R6.

## 9.3. HA-B08.01

Obsolete

This development revision was planned for a development stage of the module interconnect using a 10-pin header and serial shift register protocol. A schematic of this board revision exists in the revision control system, a prototype was constructed and tested, and in consequence the module interconnect was redesigned. No B08.01 PCBs were manufactured.

### 9.3.1. Compatibility

This revision is usable with bus logic board HA-B06.01 / HA-B06.02.

# Part III.

# Software

# 10. HA-P01, Bus device firmware template code

| |
|---|
| *HA-P01.02*: For ATxmega128-based boards, development<br>*HA-P01.01*: For ATtiny88-based boards, obsolete |

## 10.1. HA-P01.02

This code repository contains a non-functional boilerplate firmware implementation that includes basic bus protocol handling and hooks for actual functionality. It is intended to ease and speed up further development of firmware for new bus devices.

### 10.1.1. Description

The device firmware runs as an endless main loop, with I/O devices and timers raising interrupt requests (IRQs). The IRQ handlers set flags in the MCU's `GPIOR0` register, which is evaluated within the main loop and outside the interrupt context. By this procedure, normal operations are blocked for few instruction cycles only, and the actual work on the respective new interrupt is done when the interrupted operation has finished.

### 10.1.2. Interrupts

#### INT0

This I/O interrupt is used for detection of a configuration device attached to the device's I²C connector. It is configured an edge-sensitive input line, triggering on the rising edge, in `i2c.c:i2c_init()`. The internal pull-up on this line is turned off in the initialization routine, and an external light pull-down resistor is required to prevent erroneous triggering on startup and when an I²C connector is plugged in. Prototype hardware designs use a 10 kOhm value for this purpose.

#### INT1

This I/O interrupt is connected to the interrupt output of the CAN controller IC. Because the CAN controller's interrupt signal uses active-low logic, it is configured as edge-sensitive input line, triggering on the falling edge, in `mcp2515.c:canctr_init()`, but only after having initialized and configured the CAN controller. Because at that time, the CAN controller drives its interrupt output as required, there is no pull-up resistor included on this signal in the current bus device hardware designs.

#### TIMER0_COMPA

This timer compare match interrupt is used to provide the timebase for clocked activities within the bus device. It is configured in `main.c:main()` to be posted every 10 msec with sufficient accuracy for operations in the HA infrastructure. Note that if the MCU's clock frequency is chosen different from the default of 8 MHz, the prescaler and counter initialization values have to be modified accordingly.

#### TWI

The Two Wire Interface (TWI) interrupt is enabled only when a local configuration device has been detected and this device has signalled the need for I²C communications through INT0 as explained above. During I²C data transfer, this interrupt is raised by the MCU's TWI subsystem which handles the I²C communication.

In the current state of development, however, this mechanism is not fully implemented, and the interrupt handler which is run out of the interrupt context just clears the interrupt and discards any received message.

### 10.1.3. Compile-time configuration definitions

Certain preprocessor definitions influence the runtime behaviour of the firmware. Additional definitions regarding MCU I/O port usage and other hardware-dependent settings are omitted in this listing, they are handled in the section "Checklist for porting HA-P01 to new bus devices", starting on page 64.

#### MY_DTID

Defined in `main.h`, this value is the Device Type Identifier (DTID) assigned to the bus device. The registry for the values to be used here is HA-I03.

#### MY_FIRMWARE_IDX

(main.h) This is an integer value assigned to a specific revision of device firmware, and used to identify the level of supported operations and upgrade status to the HACS to free it from parsing the string firmware revision identifier. This index value may be incremented independently from the part number revision indicator assigned for bus device firmware. Is is recommended that a separate registry of index values is kept, for example by using appropriately named tags in a source code revision control system.

#### DEFAULT_CANADDR

(main.h) If the ACAA mechanism is not used by a device, which is the case at least during the development state, this value is used as the Device Identifier until another DID is configured through the I²C configuration interface.

#### TMR_HANDLE_DEVICE_Z, TMR_LED_BLINK_Z, TMR_CAN_ONBUS_Z, TMR_CAN_ONBUS_SLOW_Z

(main.c) These values set the frequency of certain periodic operations executed during the main loop. All these defines are given in centiseconds, that is, in multiples of 0.01 seconds.

The HANDLE_DEVICE define controls the execution of device_impl.c:device_handle(), responsible for the device functionality, as will explained further below in the operational documentation. The default value is 1, meaning the device handler will be called every 0.01 seconds (10 msec).

LED_BLINK is used during development and can cause a diagnostic LED at the frequency defined here. Default value is 50, giving a 2 Hz blink when the main loop is running and timed operations are executed as desired.

The combination of ONBUS and ONBUS_SLOW controls the sending of SYSM_LO_C, CI=ONBUS messages by the device. With the default values of ONBUS=100 and ONBUS_SLOW=240, the respective function is executed once every four minutes.

#### MCU, F_CPU

(Makefile:CDEFS) These mandatory Makefile variables define the MCU type in use and its operational clock frequency. These are used by both the avr-gcc compiler and firmware-internal calculations at compile-time. Their default values are attiny88 and 8000000 (8 MHz). Using other MCUs and other frequencies may require more porting work than documented here.

#### CLOCKDOWN

(`Makefile:CDEFS`) If defined, this flag enables the MCU clock being set to a lower frequency during idle times, being turned up again once any of the configured interrupts occurs. This can reduce power consumption of the device, at the cost of slightly longer response times to any external events. This feature is deactivated by default.

#### NOCAN

(`Makefile:CDEFS`) If defined, this flag makes the firmware configure the CAN controller in internal loopback mode. In this configuration, development and test data can be sent without actual bus attachment.

#### DEVEL

(`Makefile:CDEFS`) If defined, the firmware will send numeric values over the MCU's SPI interface at certain checkpoints in the code without asserting any chip select signal. These checkpoint values are intended to be sniffed of the SPI bus by an oscilloscope or logic analyzer and used for development or debugging purposes. This functionality is implemented in the `spi.h:SPIDBG()` macro.

### 10.1.4. Global variables

#### GPIOR0

This global register is used for storing interrupt flags: The interrupt service routines set a specific flag in this register, which is evaluated and acted upon in the main loop. The numeric values of these flags are defined in `main.h`. As the register is eight bit wide, eight flags can be defined to be used here, out of which four are currently in use.

#### GPIOR1

This global register is used for other globally available program state, also defined in `main.h`. Currently, two out of eight possible flag values are used.

#### versid

`const char[] versid`, stored in program memory. This is a NULL-terminated string constant consisting of colon-separated fields: Firmware author's email address, hardware part number including revision, firmware part number including

revision, and a stability indicator. The stability indicator is one of the following: dev for a development release, prd for a production release. Example:

Listing 10.1: Example version identifier string from bus device

```
"nz@thiemo.net:HA-B06:HA-P05.01:dev\0"
```

### can_stat

`uint8_t can_stat`, stored in RAM. This variable holds the result of the CAN initialization function. The value 0 means no error has occurred, the value 42 is stored before attempting to initialize the controller, and any other value indicates an error. As this variable is currently not evaluated anywhere in the code, it might be removed in a future revision of this code.

### cfg

`struct cfgmem cfg`, stored in RAM, backing store in EEPROM. This structure is used to globally manage data to be stored in and retrieved from non-volatile memory. This structure is defined in `cfgmem.h` and contains:

Listing 10.2: `struct cfgmem` definition

```
typedef struct cfgmem {
    uint8_t magic;
    uint8_t devid;
    uint8_t local_direct[5][3];
} cfgmem_t;
```

The functions to read and write this variable from and to EEPROM storage are declared also in cfgmem.h, and implemented in cfgmem.c.

`rd_cfg(cfgmem_t *cfg)` and `wr_cfg(cfgmem_t *cfg)` take a pointer to the cfg structure and update or save its contents. Their return value is EOK on success, ENOCFG if the read or write operation has failed. The EOK/ENOCFG numeric values are defined in retcodes.h.

The initial reading of EEPROM configuration contents occurs during the initialization steps in `main.c:main()`. After the contents of the in-RAM structure have changed, it is suggested to write the data to EEPROM to prevent data loss in case of unexpected power loss. However, note that the current implementation does not implement any wear leveling on the EEPROM hardware.

`magic` is written to the constant value 0x42 by `wr_cfg()` and serves as indication whether the configuration has been saved before or the EEPROM is in its default, unprogrammed state.

`devid` is used to store the CAN bus device identifier (DID), as configured or assigned through ACAA.

`local_direct` contains the configuration for Direct Event Actions as received from the HA Control Station. These are used to directly react to events by changing actor state, independently of HACS management.

### statcnt

`volatile stat_t statcnt`, stored in RAM. This structure contains counters for buffer overflow conditions in the bus device firmware and error conditions reported from the CAN controller:

Listing 10.3: `stat_t` definition

```
typedef struct {
    uint8_t ovrfl0;
    uint8_t ovrfl1;
    uint8_t ovrfl_can_rx0;
    uint8_t ovrfl_can_rx1;
    uint8_t errcnt_can_psv;
    uint8_t errcnt_can_off;
} stat_t;
```

`ovrfl0` and `ovrfl1` are incremented when the interrupt service routine for IRQ0 and IRQ1, respectively, are entered with their activity flag in GPIOR0 already set. This indicates that the main loop has not reached the actual non-interrupt handlers for these IRQs when the next IRQ of the same type was posted. This evaluation takes place within the respective interrupt service routines in main.c.

`ovrfl_can_rx0` and `ovrfl_can_rx1` are incremented by `mcp2515.c:canctr_error_counters()` when a receive buffer overflow condition has been reported by the CAN controller. This function is called from within the endless main loop after each handling of a CAN controller interrupt flag. These overflow errors occur when new CAN datagrams addressed to the bus device are received from the bus at a faster rate than the MCU under firmware control handles them and frees the CAN controller's receive buffers.

Note that the bus device firmware daisy-chains the two available receive buffers, so one message can be stored while one is being processed. The CAN controller includes a third buffer into which a datagram is assembled while being received - it is at the end of the reception process that a free receive buffer is required, and, if not available, an overflow error is flagged.

`errcnt_can_psv` and `errcnt_can_off` are counters of *error-passive* and *error-off* conditions reported by the CAN controller. These are currently not evaluated, and will read as 0 at all times.

## 10.1.5. Operation

### Main loop

After reset and entering the `main()` routine, the MCU's input and output lines and the device hardware are initialized. The first subsystem to be set up is the SPI communication interface, because in development builds, this is used for the transmission of checkpoint indicators to be observed with lab equipment. A list of checkpoint values is included in this documentation. After that, the other subsystems are activated.

The firwmare reads the configuration space from EEPROM and checks for the *magic number* to see whether it has been written before. If this is not the case, the compile-time default CAN device identifier is used, and the configuration EEPROM is initialized with this value.

After a CAN DID has become known by the previous steps, the CAN controller's address filters are configured to match that identifier in the respective bits of the CAN message identifier. Note that in the current implementation, one ot the two available receive buffers is disabled for message reception and is used as an overflow buffer for the other, active one. Once Direct Event Actions (DEA) are to be supported, this mechanism has to be changed to receive datagrams that need to be evaluated by the DEA functions.

According to the HA-CAN Higher Level Protocol, a `SYSM_L0_C, CI=D_RESET` notificatoin is then sent, followed by an initial `SYSM_L0_C, CI=ONBUS` mesage.

After this, interrupts are enabled, and the endless main loop is entered. The main loop sequentially checks GPIOR0 for flags set by the interrupt service routines and, if any is active, calls an appropriate worker function before atomically unsetting the handled flag from the status register. Atomic operation here is achieved by disabling interrupts, doing the read and write operation on the register, and enabling interrupts again. Note that the ISRs do not take such precautions, because the supported MCU architecture handles only one interrupt at a time in the configuration used by this firmware, and therefore bears no risk of concurrent register access.

If no status flag is left active in GPIOR0 after one execution of the main loop, the MCU's IDLE state is entered. If the compile-time configuration CLOCKDOWN is enabled, the clock frequency is reduced before entering IDLE mode.

The next intterupt (external, SPI, I²C, or timer) will end IDLE mode and, if enabled, restore the clock frequency to normal.

### CAN message handling

If `ST_FIRED_INT1` is set in the GPIOR0 register in the main loop, this signals that an interrupt request has been posted by the CAN controller. The main loop there-

fore calls the `mcp2515.c:canctr_get_message()` function, reading the received message into a `can_t can_msg`. This is defined as follows in `can_proto.h`:

Listing 10.4: `can_t` definition

```
typedef struct can {
    uint16_t id;            /* 5 bit bus number, 11 bit canid */
    struct {
        uint8_t rtr : 1;    /* remote transmit request? */
    } flags;
    uint8_t length;
    uint8_t data[8];
} can_t;
```

### 10.1.6. Checkpoint values on the SPI in development builds

The following checkpoint values are included in the template code, listed in chronological order for events before entering the main loop and enabling interrupts, in numerical order for checkpoints reached afterwards. The boundary between the two cases is marked with a thicker line in the table.

| Value | Subsystem | File:Function | Description |
|---|---|---|---|
| 0x42 | Initialization | `main.c:main()` | SPI has been initialized |
| 0x43 | Initialization | `main.c:main()` | All subsystems but CAN have been initialized |
| 0x44 | Initialization | `main.c:main()` | CAN has been initialized |
| 0x45 | Initialization | `main.c:main()` | Endless main loop is about to be entered |
| 0x50 | Main loop | `main.c:main()` | Timer 0 (10 msec timer) has fired |
| 0x51 | Main loop | `main.c:main()` | `handle_device()` is about to be called |

### 10.1.7. Checklist for porting HA-P01 to new bus devices

The firmware for the 16-port digital input bus device is the first bus device firmware being implemented and tested. It can serve as a template for future bus device implementations, and requires some adjustments in the code when porting it. These necessary adjustments are listed here for future reference.

#### Administrative

Assign a new device type ID in HA-I03, see the references there for other places that need to be updated for changes to that document.

### Files

*device_impl.[hc]*

Adjust defines `MY_IORANGE_INPUT` and `MY_IORANGE_OUTPUT` to the number of input and output channels, respectively.

Adjust defines or implement functions `device_mgt_range_lo()` and `device_mgt_range_hi()` to return the minimum and maximum values for multi-value inputs and outputs for a given channel.

`device_init()`: This function is called once after system reset, after the other sub-systems have been initialized. Implement device-specific initialization functions here.

`device_handle()`: This function provides a comfortable entry point for the device-specific functionality implementation. It is called frequently by the rest of the program, and should be used to do any local I/O operations, and sending of CAN datagrams when required.

`device_can_process_msg()`: This function is called by the CAN receive subsystem when a device-specific message is received from the bus.

*leds.h*

Adjust defines for LED I/O assignments.

Adjust defines for available LEDs.

*leds.c*

`led_init()`: Implement functionality for available LEDs.

`led_on()`: Implement functionality for available LEDs.

`led_off()`: Implement functionality for available LEDs.

*main.c*

Adjust the versid string to include the correct part and revision identifiers.

*main.h*

Adjust define for device type ID from HA-I03.

*mcp2515.c*

Depending on controller type, adjust register names for interrupt control.

*spi.h*

Adjust defines for SPI I/O assignments.

*spi.c*

Depending on controller type, adjust register names for SPI control.

### CAN controller driver interface

If a different CAN controller shall be used, the `mcp2515.[hc]` files can be replaced by another driver implementation. This change needs to be reflected in all include sections in the C sources files used.

Because the CAN higher layer protocol (HLP) is a relatively large chunk of code that can be reused in other device implementations, it is suggested to either use one such HLP implementation by copying it into new device firmware projects, or produce a proper library against which new firmware is linked. The reference implementation contains the CAN HLP in `can_proto.[hc]`, as described.

The CAN controller driver implementation needs to provide a set of functions that are called by the rest of the CAN handling code. Note that all of these could be either implemented as functions or macros, with the usual tradeoff between program memory and stack area footprint.

*`uint8_t canctr_init(void)`*

Reset and initialize the CAN controller. Return “0” on success, and any other unsigned 8-bit integer on error.

*`uint8_t canctr_send_message(const can_t *msg)`*

Submit a CAN message to the controller for transmission, and initiate transmission. Return one of the return codes defined in `retcodes.h`.

*`uint8_t canctr_get_message(can_t *msg)`*

Copy a CAN message from the controller out into the provided message buffer pointed to by msg. Return one of the return codes defined in `retcodes.h`.

*`uint8_t canctr_program_filter(struct cfgmem *cfg)`*

Program the controller’s message filters (if supported) with the information found in the controller configuration pointed to by cfg. Return one of the return codes defined in `retcodes.h`.

*uint8_t canctr_error_counters(volatile stat_t *statcnt)*

Read the controller's error states and, if applicable, implement the error counters contained in the statistics data structure pointed to by statcnt. Return one of the return codes defined in `retcodes.h`.

*uint8_t canctr_get_rxerrors(void)*

Read and return the receive error counter from the controller.

In the reference implementation, this is implemented as a macro, because the actual controller interaction is already implemented in `mcp2515_read_register()`.

*uint8_t canctr_get_txerrors(void)*

Read and return the transmit error counter from the controller.

In the reference implementation, this is implemented as a macro, because the actual controller interaction is already implemented in `mcp2515_read_register()`.

*uint8_t canctr_get_txwarn(void)*

Find out if the controller's transmit warning level is reached. Return "0" if not, any other value if it is.

In the reference implementation, this is implemented as a macro, because the actual controller interaction is already implemented in `mcp2515_read_register()`.

*uint8_t canctr_get_rxwarn(void)*

Find out if the controller's receive warning level is reached. Return "0" if not, any other value if it is.

In the reference implementation, this is implemented as a macro, because the actual controller interaction is already implemented in `mcp2515_read_register()`.

# 11. HA-P03, Software on the Control System

| |
|---|
| *HA-P03.02*: Development<br>*HA-P03.01*: Production |

## 11.1. Control System configuration

The configuration of managed devices, actions executed in response to events, and automatic actions to be triggered depending on certain input values is configured on the control system.

For the overall configuration of basic parameters, there is one main configuration file, `ha.conf`, which among other settings contains pointers to more specific configuration files defining the devices, actions, device groups, and macro functionality. While the main `ha.conf` is a "INI-file" format, the specific configuration files are written in a specific XML format to allow for greater flexibility.

### 11.1.1. Main configuration file, ha.conf

The HA software stack on the Control System is configured through a central configuration file, `ha.conf`. By default, it is located in and searched for in the directory `/usr/local/etc/`.

The HA configuration file is written in the so-called "INI-file" format and contains sections for the subsystems to configure. Those sections and the supported configuration options are documented in the respective software sections below, but three sections are evaluated by all parts of the software stack, and influence the behaviour of the software as a whole. These sections and their default values are as follows:

Listing 11.1: Sample ha.conf

```
[DEFAULT]
haroot = /usr/local

[system]
log = file
logfile = /var/log/ha.log
```

```
confdir = %(haroot)s/etc/ha
libdir = %(haroot)s/lib/ha

[files]
actions = actions.xml
devices = devices.xml
groups = groups.xml
macros = macros.xml
```

### 11.1.2. Device configuration

This configuration file defines the devices managed by the HomeAutomation system. While several different components of the software stack require different information about the managed components, it is all defined in the respective sections of this configuration file.

This XML file contains a sequence of blocks, each defining several aspects of one device.

#### Device block

The main object of this configuration is a “device”, which is an instance of one device type, and to which a unique identifier string is assigned. Through this identifier string, the device is addressed in other configuration files and internally within the control software.

The XML configuration looks as follows:

```
<device id="[deviceid]" type="[device_type]">
  [ ... ]
</device>
```

`[deviceid]` is freely assignable, but should not be chosen too long for reasons of readability. There are other ways to configure human-readable longer description strings which are explained further below.

`[device_type]` must be one of the supported device types for which a device type class implementation exists in the `ha_common` directory. The available classes are documented in the `ha_common` section, beginning on page 72.*XXX* To be documented.

XXX

`[ ... ]` is shown here just to indicate that within the device definition block, several other configuration directives are expected, as will be explained in the following sections.

### 11.1.3. Action configuration

*XXX* XXX

### 11.1.4. Device group configuration

To ease configuration of actions to be applied to several devices at once, devices can be grouped and then addressed in action configurations using the group identifier assigned when defining the group. Groups can also be configured to be members of other groups, note, however, that circular group definitions are not allowed and will be ignored when the group configuration file is evaluated. *XXX* XXX

### 11.1.5. Macro configuration

Just as device groups are collections of devices joined under a common designator ID, macros are essentially collections of actions that are executed sequentially by the control software. However, the HA macro definitions can also contain control statements, for example to delay execution for a given time period, or take different actions depending on the value available in state variables (as kept by `ha_stated`.

*XXX* XXX

## 11.2. Address calculator utility

| *ha_calc*: Production |
|---|

This helper application calculates bus numbers, device addresses and, if required, message type identifiers from a given numeric CAN ID.

Listing 11.2: Sample `ha_calc` usage

```
> ha_calc
Usage:
  ha_calc CANID
  ha_calc BUS DEVID [MTID]
> ha_calc 771
CANID 771: bus 0, devid 3, mtid 6
> ha_calc 0 3
CANID 3: bus 0, devid 3, mtid 0
```

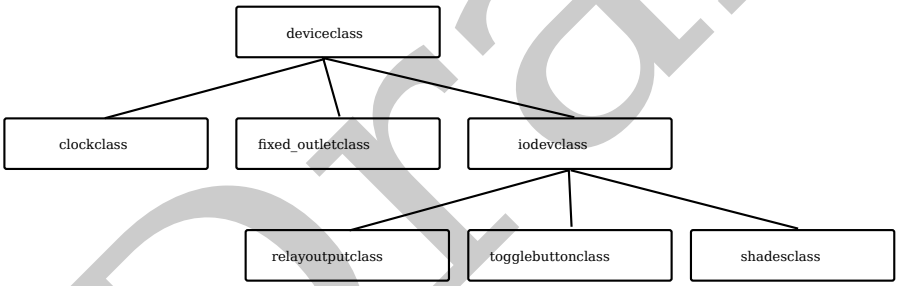
## 11.3. ha_common

| *ha_common*: Production |
|---|

The `ha_common` directory contains Python module files that are used by other scripts on the HACS. Because they can probably be useful to more than one other program, they are stored in a commonly accessible library location in the HACS, and are written to be ignorant to the script that is actually using them. This section describes the contents of the `ha_common` directory, and documents the Application Programmer Interface (API) to these modules.

### 11.3.1. Device classes

CAN bus devices are represented by an object-oriented abstraction layer, which is their respective instance of a `Device` class, tailored to the specific device type. All `Device` classes inherit basic functionality from a basic class, `deviceclass.Device`. The following child classes are currently implemented:




The `Device` objects provide a set of methods to interact with the device representations. The following sections document the interfaces to software using these objects.

XXX *XXX*API documentation needs to be imported from the `epydoc`-generated documentation in the `ha-p03` repository, or a reference to that needs to be added here.

## 11.4. ha_control

| *ha_control*: Production |
|---|

ha_control receives events from various HomeAutomation subsystems and converts them into action commands according to rules configured by the user, either directly or by executing macro threads that in turn emit action commands into the HA software stack or the HA-CAN bus.

This program also provides an XML-RPC interface for user programs to interact and exchange status information with bus devices and the HomeAutomation software infrastructure.

Additionally, ha_control monitors the system clock, emits time-based events, and provides bus devices with time of day information.

## 11.4.1. Description

*XXX* Document device abstraction layer, actions, events, threads interaction, external interfaces. Reference auto-generated API documentation for further details. XXX

## 11.4.2. Configuration

### Program behaviour

The overall behaviour of the ha_control daemon software is configured in two sections of the ha.conf configuration file.

Listing 11.3: Sample ha.conf entries for ha_control

```
[ha_control]
cleanup_interval = 15
debug = yes
eventfilter = SYSM_LO_M:*
filterdefault = pass
host = localhost
port = 3238
tick_seconds = 5
timer_check_interval = 60

[values]
multiclick_ms = 400
```

### Device configuration

Devices, that is event sources and action command recipients, are configured within the <device> elements in the devices.xml file, unless configured otherwise through the respective entry in ha.conf.

# 11.5. ha_iod

| *ha_iod*: Production |
|---|

This daemon opens and maintains the serial connection to the CAN bus interface, it buffers and serializes all bus transmissions (incoming and outgoing). `ha_iod` provides an interface to the rest of the software stack by means of TCP connections.

`ha_iod` contains little logic besides the device management and message buffering, messages passed to it must be preformatted for more or less direct transmission to the bus interface, and messages being passed to the software stack by it are little more than raw CAN datagrams encoded for serial transfer.

The protocol between `ha_iod` and its TCP clients is defined as HA-I04 (page 103).

The protocol between `ha_iod` and the HA-B02 USB/CAN interface is defined as HA-I05 (page 107).

## 11.5.1. Usage

`ha_iod` is intended to be started at system start of the HA control computer by means of the operating system's startup mechanism.

When called with the `-h` option, it prints a short usage information, giving possible parameters and their default values:

Listing 11.4: Output of `ha_iod -h`

```
> ha_iod –h
Usage: ha_iod [option] [...]
Options:
  –d usb_device  USB serial device file ("")
  –e sock_file   Emulator AF_UNIX socket file ("")
  –p tcp_port    TCP port to listen for connections (3236)
  –l             Enable rate checking on USB
  –s usb_speed   line speed on USB serial device (115200)
```

Either the USB device that has been assigned to the CAN/USB interface or a socket file name must be provided for operation, otherwise the program will terminate with an error message and exit code `EX_OSERR`. If both parameters are provided, the USB device is opened and the emulator socket is ignored.

The rate checking mechanism can limit the number of both datagrams and bytes received from and sent to a CAN bus attached to the CAN/USB interface. On reception and transmission of data through any CAN bus, bytes and frames are counted by `ha_iod`, and the respective rate per second is calculated. If this rate exceeds a limit defined at compile-time (`USB_MAXFPS` and `USB_MAXBPS` in `usb.h`), an error message is logged and both CAN controllers are reset.

### 11.5.2. Configuration

Listing 11.5: Sample `ha_iod` configuration

```
[ha_iod]
host = localhost
linespeed = 115200
port = 3236
```

### 11.5.3. Maintenance

#### CAN/USB interface identification string

`ha_iod` has a compiled-in list of CAN/USB interface identification strings and queries that string upon startup. Depending on the result, flags can be set to control the software's behaviour when interacting with the interface hardware.

The list of ID strings and the applicable flags is contained in `udev_versiontab` in file `usb.c`. This file needs to be updated when a new ID string is reported from a CAN/USB interface that requires specific handling from `ha_iod`.

#### CAN device type identifier (DTID)

A list of assigned DTIDs is present in `can_dtidtab` in file `can_dtid.c` through which necessary special handling can be configured by means of DTID-specific flags, and where identification strings are assigned to DTIDs that can be reported to TCP client software when retrieving a device list.

This DTID list needs to be updated when new device type identifiers are assigned.

#### CAN message type identifier (MTID)

Symbolic names for MTIDs that are used in the program code are defined in the file `can_proto.h`.

This MTID list needs to be updated when MTID definitions are changed or new MTIDs assigned. SYSM command identifiers (CIs) are defined in this file as well, so it also needs to be updated when CI assignments are changed or extended.

## 11.6. hashell

| *`hashell`*: Development |
|---|

The command line interface to the rest of the software stack is used in development of software and hardware, and may later serve as debugging interface and to access certain administrative or low-level functionality — or just to control the system from non-graphical terminals.

The `hashell` currently interacts directly with the configuration files, and opens its own connection to `ha_iod`. In the future course of development, an XML-RPC interface of `ha_control` is planned to be used for at least many, if not all, of the interactions with CAN devices and state.

```
hashell [–c config]
```

The `-c` option can be used to have the sofware read a specific configuration file instead of `/usr/local/etc/ha.conf`.

### 11.6.1. Commands

On the shell command line, `help <command>` can be used to get more information about a command.

Listing 11.6: Command reference for `hashell`

```
canlist – Prints a list of CAN devices known to ha_iod.
canreset <bus> – Resets the CAN controller on bus <bus>.
confdb [subcommand] [args] – Show and manipulate contents of the configurati...
dev <addr> – Changes the current device to the device ID <addr>.
event subcommand [args] – Generate events and post them to ha_control
exit – Gracefully terminates hashell.
help [topic] – Provides some assistance on hashell's commands and syntax.
info – Prints device information.
ls – An alias for "info"
pwr <bus> <state> – Switch a CAN bus power supply on or off
repair <channel> – Declare a previously failed channel on an I/O device as r...
rly set ... | rly rq ... – Relay channel state operations –
rq <request type> [<devid>] – Send a request to the active client.
send [rtr] <mtid> [<data>] [...] – Sends a CAN datagram using the current de...
state subcommand [args] – Interact with the statekeeping subsystem
threads [–l] – List ha_control's running threads
```

Subcommands for the `event` command:

```
    macro <name>                                    Start macro <name>
    send <id> <bus> <devid> [data1] .. [data8]  Create an event and post it
                                                    to ha_control
```

Subcommands for the `state` command:

```
    ls                  Get a list of all (public) state variables
    save                Save current state to stable storage
    tm                  Calculate when the next timer will fire
```

# 12. HA-P04, Dual CAN/USB bus interface (HA-B02) firmware

| *HA-P04.01*: Prototype |
|---|

This is the firmware for the HA-B02 USB/dual-CAN bus interface. It translates and forwards messages back and forth between the host computer and the CAN devices on internal and external buses.

# 13. HA-P05, Modular CAN-I/O bus logic board (HA-B06) firmware

| |
|---|
| *HA-P05.02*: For HA-B06.03-: Under development<br>*HA-P05.01*: For HA-B06.01-02: Prototype, obsolete |

## 13.1. HA-P05.02

### 13.1.1. Description

The firmware running on the Modular CAN-I/O Bus Logic Board (BLB) takes care of all operations required in a CAN bus device: It handles the administrative CAN message traffic, operates peripheral devices, generates the necessary messages in response to peripheral sensor readings, and manipulates actors in the physical world according to messages received from the bus. Because of the computing resources required for these activities, the microcontroller on the BLB is chosen from the relatively powerful Atxmega family, and therefore the device firmware is tailored to this MCU family.

In general, the BLB firmware operates in an endless loop, with interrupts being triggered from the CAN subsystem, internal timer interrupt sources, and I²C and UART configuration and debug interfaces present on the board.

The actual functionality of the Bus Logic Board, and therefore the functions executed by the BLB firmware, depend on the respective I/O module attached to it. Module configuration and communication parameters (in particular, the CAN device identifier) need to be configured manually before the bus device becomes operational and participates in CAN message exchange. The protocol used for configuring these parameters is described in HA-I10, starting on page 123.

### 13.1.2. Diagnostics

The firmware uses different ways to notify a user about the current state of the system. When the system is operating normally, all regular diagnostic messages are transmitted through the HA-CAN bus protocol. During startup and initialization, though, this communication channel is not yet ready to use, so other signalling mechanisms are utilized.

### LED patterns

During the early phase of system startup, there is no peripheral device configured and usable. The toggling of CAN bus connector LEDs is the first method of interaction with the physical world that is available to the firmware, so early stages of the firmware use this as a method to signal its internal state. The following table lists the LED patterns with their respective meaning in the order of program execution, as long as the main program loop has not been entered.

The LED patterns that include the `CAN_2` yellow LED blinking at 2 Hz are emitted from within the main loop, and can appear in any order.

| `CAN_1` connector | | `CAN_2` connector | | Description |
|---|---|---|---|---|
| green | yellow | green | yellow | |
| ON | OFF | OFF | OFF | MCU started, LED control signals initialized |
| ON | ON | OFF | OFF | System clock switched from 2MHz to 16MHz from 8Mhz crystal |
| OFF | ON | OFF | OFF | SPI interface initialized. SPIDBG messages are emitted through SPI |
| OFF | OFF | OFF | OFF | I²C interface initialized |
| (any) | (any) | (any) | @2 Hz | System in main loop and operating normally |
| flash | (any) | (any) | @2 Hz | Device is handling a CAN datagram |
| ON | (any) | (any) | @2 Hz | Error: Device is stuck handling a CAN datagram, reset required. |

### SPIDBG messages (DEVEL firwmare)

*Note*: This section applies only to firmware that was compiled with the DEVEL compile time option. Firmware versions not compiled with this option set do not issue diagnostic messages through SPI.

After the SPI subsystem has been initialized, the firmware emits one-byte messages through the SPI interface depending on the code location currently executed. While emitting these messages, no “Slave Select” signal is asserted, so the diagnostic data is not interpreted by any device connected to the MCU’s SPI. To access these messages, either the `SPI_MOSI/SPI_SCK` or `SPI_MOSI_5V/SPI_SCK_5V` signals on the BLB need to be probed with an oscilloscope or logic analyzer.

Because this feature is meant for development and debugging, the specific message codes and their meaning are only listed in the source code (in file debugcodes.h), and documented by their respective appearance in the code.

#### UART messages

Once the peripheral subsystems have been initalized, diagnostic output that cannot be transmitted by the regular HA-CAN protocol means is sent as ASCII character messages through the on-board UART interface. The communication parameters are 9600 bps, 8 bit, no parity, 1 stop bit. Each message is terminated by a sequence of one cursor-return and one line-feed ASCII character.

Note that the UART interface on all currently existing of HA-B06 is designed for 0-5 V signal levels. *This means the interface is not designed for RS-232 signal levels without additional circuitry!* Please check the hardware documentation (page 35) for further information on how to connect this interface.

At the time of this writing, no diagnostic UART messages are defined. Once the need arises, they will be specified in an updated revision of this document.

### 13.1.3. Firmware updates

The BLB receives and processes firmware updates through the CAN bus as described in HA-I12 (page 133). Support for this is implemented in the firmware, split into the regular application section (which handles the CAN protocol communications) and a special bootloader section, which can overwrite the normal application section memory when a new firmware image is present in the I²C EEPROM.

Software support for this protocol is currently under development.

# 14. Remote Access

Some software components are used to enable remote access to the HA systems. These run on external systems, for example a web server with connection to the Internet, and communicate with the HA-internal software through special communication channels.

This section explains such external software components, while the communication protocols are defined in the Protocols section, and the software running on the HACS itself was documented in an earlier section.

# Part IV.

# Protocols and Interfaces

# 15. HA-I01, Automatic CAN address assignment (ACAA)

| *HA-I01.01*: Under development |
|---|

## 15.1. HA-I01.01

This subsection describes the protocol used for automatic assignment of device identification numbers (CAN-IDs) to newly attached bus devices.

### 15.1.1. Mode of operation

When a HA-CAN bus device is manufactured, the device identification number (CAN-ID) of it is not yet known. For ease of use, it is desirable that this does not have to be programmed into the device manually when the device is attached, but automatically assigned by the control software. The control software keeps a list of all attached devices anyways (together with their configuration), so it can assign the next free CAN-ID to any new device that is seen. The operator can then manually configure this device as required, and at a time suiting operation's needs, while the device is already active on the bus, but not hindering other bus transactions.

For this functionality, a specific CAN-ID is defined under which the Automatic CAN Address Assignment (ACAA) datagrams are transmitted. This CAN-ID is reserved, and will not be used for regular bus device communications. Only ACAA traffic is permitted using this CAN-ID. The ACAA process is designed to be fast enough for assigning a "real" CAN-ID to a new device on the bus before another new device is added in all practical situations, given the intent of this whole project.

### 15.1.2. Protocol description

The ACAA protocol consists of two steps. First, a request for a CAN address is sent to the bus using the specified ACAA CAN ID, and under this CAN ID the address is transmitted to the client. The requests includes a unique identifier of

the new device hardware, for example the MCU's controller identifier as pre-programmed by the manufacturer. Users need to ensure that this is indeed a unique identifier in one installation at a given time. Both request and response datagrams include the new device's Device Type Identifier (DTID).

Afterwards, to ensure correct transmission and receipt of the assigned address, a check transmission is sent to the client and acknowledged by it. Only after successful reception of this check packet is the client allowed to store the new address as its fixed CAN identity and send the respective acknowledgement datagram. Only after receipt of the check acknowledgment is the ACAA server allowed to permanently register this CAN ID for the new device.

If assignment and verification fails in any step, the new client device must start the process from the beginning. The server is in this case expected to hand out only a limited number of addresses through ACAA, and, if assignment repeatedly fails, stop responding to ACAA queries and request operator intervention.

If the assignment process failes from ACAA manager viewpoint after handing out the new CAN ID (i.e., after it has sent frame number 2 in the following table), it must mark the respective ID as blocked, and request operator intervention before marking it as clear to hand out again.

### 15.1.3. Device ID

Device ID 121 has been assigned and reserved for the ACAA traffic.

### 15.1.4. Protocol message flow

XXX This table is broken.*XXX*

| Frame | New device (client) | | | | ↔ | ACAA manager (server) | | | | Comments |
|---|---|---|---|---|---|---|---|---|---|---|
| | MTID | CANID | Length | Data | | MTID | CANID | Length | Data | |
| 1 | 14 | 121 | 8 | 1: 70 2: DTID 3: UID1 4: UID2 5: UID3 6: UID4 7: UID5 8: UID6 | → | | | | | Address request for device type DTID. The UIDn fields are unique hardware identifier bytes. If less UID bytes are supported, the remainint required packet's data bytes are padded with 0x00 to give a full datagram with 8 data bytes. |
| 2 | | | | | ← | 14 | 121 | 3 | 1: 71 2: DTID 3: CANID | Assignment of bus address CANID |
| 3 | | | | | ← | 14 | CANID | 1 | 1: 72 | Request for confirmation of new address |
| 4 | 14 | CANID | 1 | 1: 73 | → | | | | | Confirmation of successful configuration |

The ACAA manager should wait 20 to 30 ms between step 2 and 3 to allow for client's adjustment of address configuration.

# 16. HA-I02, Higher Layer CAN Protocol

| *HA-I02.01*: Production |
|---|

## 16.1. HA-I02.01

This protocol defines the format and content of messages exchanged on the CAN bus. It serves as a *higher layer protocol* as understood in the CAN specification (ISO 11898).

*XXX* Bus speed should be defined somewhere. Probably not in the HLP, though. XXX

### 16.1.1. Message identifier

In general, the *standard addressing* scheme with 11-bit identifiers is used. The first 4 bits (bit 10..7) are used to identify the message type (Message Type Identifier, MTID), the last 7 bits (bit 6..0) are used to uniquely identify a device on the bus (Device Identifier, DID). Five additional bits (11..15) are used to identify different CAN buses within the host controller software.

| 15 | 14 | 13 | 12 | 11 | 10 | 09 | 08 | 07 | 06 | 05 | 04 | 03 | 02 | 01 | 00 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bus ID | | | | | Message Type Identifier | | | | Device Identifier | | | | | | |

This allows for 15 message types to be transmitted (see below for explanation and assignment) and 128 devices on the bus. Certain device addresses are reserved for future use, so this will lead to 120 uniquely addressable devices on the CAN bus. A multi-bus configuration with routing capabilities is currently not implemented, but taken into account in this protocol definition.

### 16.1.2. Message Type Identifier

#### Overview

The following Message Type Identifiers (MTIDs) are defined.

| MTID | Short name | Class | Description |
|---|---|---|---|
| 00 | | | (reserved) |
| 01 | SYSM_HI_M | Command | System management message, high priority, request |
| 02 | SYSM_HI_C | Event | System management message, high priority, response |
| 03 | | | (reserved) |
| 04 | | | (reserved) |
| 05 | INDIGPULS | Event | Input status sense, digital, pulse |
| 06 | OUTDIG | Command | Output status command, digital |
| 07 | OUTVAL | Command | Output status command, value |
| 08 | | | (reserved) |
| 09 | INVAL | Event | Input status sense, value |
| 10 | INDIGPERM | Event | Input status sense, digital, permanent |
| 11 | | | (reserved) |
| 12 | SYSM_LO_M | Command | System management message, low priority, request |
| 13 | SYSM_LO_C | Event | System management message, low priority, response |
| 14 | BULKDATA | Variable | Bulk data transfer protocols |
| 15 | | | (invalid) |

Blocking MTID 15 from use fulfils the CAN standard's requirement that not all first 7 bits may be recessive (logical 1).

The SYSM_* MTIDs support a request-response message flow where the control station requests for example status data from a bus device, and the device responds. This requires different MTIDs for both possible directions of message flow: Because the actual payload is encoded in the message's data bytes, such different payload is not considered by the CAN protocol's bus arbitration mechanism. Therefore, with identical MTIDs, datagram collisions on the bus would be possible if both devices encoded the same values for client device address (either as recipient or sender of the message) and MTID. This is undesired behaviour and mitigated by the different message type identifiers (which are encoded in the actual on-bus CAN message identifier).

### Message Type Details

The following sections will explain the data transmitted in the CAN message datagrams. Data content depends on the message type and thus can be identified by a bus member by first classifying the datagram according to its MTID and then parsing the message payload.

*MTID 01/02, SYSM_HI_M/SYSM_HI_C: System management message, high priority*

*Currently unused, to be defined.*

*MTID 05, INDIGPULS: Input status sense, digital, pulse*

This message signals the detection of a pulse on a digital pulse input. The message is generated when an input known to be a digital pulse input is activated. The message is not repeated when the input stays active for any amount of time. After the input goes passive (is deactivated) for a duration depending on the input and bus device specification and implementation, it can be triggered again, causing this message to be sent again.

An example might be a pushbutton having been pushed. When the user presses the button, this message is sent. When the user holds down the button after pressing it, no further action is taken. The next message will be sent when the user releases the button and it is pressed again.

Double-click functionality is currently only specified as a concept to be supported by the respective devices, implementation details are yet to be defined.

Data fields for MTID 05 INDIGPULS (Input sense, digital, pulse):

data[0] channel number

| | | |
|---|---|---|
| data[1] flags: | 00 | no flags |
| | 01 | double-click |
| | 02 | long press |

Total data bytes: 2

*MTID 06, OUTDIG: Output status command, digital*

This message commands an output device to set one digital output to a certain state.

The output pulse commands are executed only when the output is already set to the respective other stable state when the command arrives. For example, when an output is switched off, an output pulse low command will be ignored. The duration of an output pulse is application-specific and depends on the bus device that is addressed. Commanding the state “toggle” makes the recipient actor read the current I/O port state and switch the port to the opposite state. The new state is reported as usual through a SYSM_LO_C, CI=S_IOSTATE message.

Data fields for MTID 06 OUTDIG (Output status command, digital):

data[0] channel number

| | | |
|---|---|---|
| `data[1]` port state: | 00 | off |
| | 01 | on |
| | 03 | toggle |
| | 06 | pulse on |
| | 10 | pulse off |
| | 21 | failed (report only) |

Total data bytes: 2

All state codes are decimal. States marked “report only” are not to be set by CAN datagrams, but to be included in channel status reports in other message types.

*MTID 07, `OUTVAL`: Output status command, value*

This message commands an output device to set one multi-value (e.g. analog) output to a certain value. As one output channel can support multiple values at once (for example, a light source that can be configured with separate color values for red, green, blue, and brightness), up to four value bytes (data bytes 1 to 4) can be transmitted. The CAN datagram’s *data length code* indicates the actual number of data bytes in a transmission.

Value can be 0..255, though not all output devices need to implement the full range of values. See MTID 13, `D_RANGE` command, for means of determining the available value ranges per output device.

Data fields for `MTID 07 OUTVAL` (Output status command, value):

`data[0]` channel number

`data[1]` value (0..255)

`data[2]` value (0..255)

`data[3]` value (0..255)

`data[4]` value (0..255)

Total data bytes: 2..5

*MTID 09, `INVAL`: Input status sense, value*

This message signals a changed value on a multi-value input device, e.g. an analog or multi-step input.

Value can be 0..255, though not all input devices need to implement the full range of values. See MTID 13, `D_RANGE` command, for means of determining the available value ranges per input device.

Data fields for `MTID 09 INVAL` (Input status sense, value):

`data[0]` channel number

`data[1]` value (0..255)

Total data bytes: 2

*MTID 10, INDIGPERM: Input status sense, digital, permanent*

This message signals a changed state in a latching digital input device, e.g. a classic light switch.

Data fields for MTID 10 INDIGPERM (Input status sense, digital, permanent):

data[0] channel number

| | | |
|---|---|---|
| data[1] state: | 00 | off |
| | 01 | on |

Total data bytes: 2

If the input device does not provide a distinguished “on” or “off” state, the two available states can be freely assigned.

*MTID 12/13, SYSM_LO_M/SYSM_LO_C: System management message, low priority*

This message type can carry several types of low priority bus and system management traffic. The specific meaning of a message is encoded in the Command Identifier (CI), which is always the first data byte in the CAN datagram payload. The overall data length depends on the actually transmitted command.

Some SYSM_LO_M commands require information included in the request datagram to further identify the data being requested, for example an I/O channel number. Bus devices must ignore data in request packets that they are supposed to fill in, though the requesting device should set these payload bytes to 00.

MTID 13 response datagrams must also contain the identification data as required in the request packet above. If a requested data is not supported by a bus device, the device must respond with an MTID 13, CI 073, M_UNSUP, message.

000-029 Bus and device administration

CI 000

*(reserved)*

CI 001: ONBUS

data[0] 001

data[1] DTID

data[2] firmware rev index

Total data bytes: 3

`CI 002: OFFBUS`

`data[0] 002`

Total data bytes: 1

`CI 003: SLEEPING`

`data[0] 003`

Total data bytes: 1

`CI 004: E_COUNTER`

`data[0] 004`

`data[1]` RXERR value

`data[2]` TXERR value

Total data bytes: 3

`CI 005: E_OFLOW`

`data[0] 005`

`data[1]` RX0 overflow counter

`data[2]` RX1 overflow counter

Total data bytes: 3

`CI 006: E_WARN`

`data[0] 006`

`data[1]` Warning level on counter (1=TXERR, 2=RXERR, 3=both)

Total data bytes: 2

`CI 007: E_ERRORS`

`data[0] 007`

`data[1]` error-passive counter

`data[2]` error-off counter

Total data bytes: 3

030-039 Device failures and status notices

CI 030: S_FAIL

data[0] 030

data[1] input number

Total data bytes: 2

CI 031: S_DEGRADED

data[0] 031

data[1] input number

Total data bytes: 2

CI 032: S_OKAY

data[0] 032

data[1] input number

Total data bytes: 2

CI 033: S_OFAIL

data[0] 033

data[1] output number

Total data bytes: 2

CI 034: S_ODEGRAD

data[0] 034

data[1] output number

Total data bytes: 2

CI 035: S_OOKAY

data[0] 035

data[1] output number

Total data bytes: 2

CI 036: S_IOSTATE

data[0] 036

data[1] channel number

data[2] state

data[3] switch toggle time in deciseconds

Total data bytes: 4

040-049 Bus device management

CI 040: D_RANGE

data[0] 040

data[1] channel number

data[2] minimum value

data[3] maximum value

Total data bytes: 4

CI 041: D_NAMERQ

data[0] 041

data[1] device ID

data[2] channel number

Total data bytes: 3

CI 042: D_NAMERP

data[0] 042

data[1-6] (device name, six ASCII chars, left-padded with space characters)

Total data bytes: 7

CI 043: D_RESET

data[0] 043

| | | |
|---|---|---|
| `data[1]` flags, one of the following: | 00 | no further information available |
| | 01 | Power-on reset |
| | 02 | Power-on reset after unexpected power failure |
| | 10 | Watchdog timeout |
| | 11 | Watchdog kicked too fast |
| | 12 | Stack overflow |
| | 13 | Firmware initiated reset |
| | 14 | PDI reset |

Total data bytes: 2

`CI 044`

*(unassigned)*

`CI 045: D_IORANGE`

`data[0]` `045`

`data[1]` input channel count

`data[2]` output channel count

Total data bytes: 3

`CI 060: I_WTIME`

`data[0]` `060`

`data[1]` day of week (0=Sunday)

`data[2]` century

`data[3]` year in century

`data[4]` month

`data[5]` day of month

`data[6]` hour (24h format)

`data[7]` minute

Total data bytes: 8

`CI 070: M_ACAARQ`

`data[0]` `070`

`data[1]` Device Type Identifier

Total data bytes: 2

CI 071: M_ACAASG

data[0] 071

data[1] Device Type Identifier from M_ACAARQ

data[2] Newly assigned Device ID

Total data bytes: 3

CI 072: M_ACAACK

data[0] 072

Total data bytes: 1

CI 073: M_UNSUP

data[0] 073

Total data bytes: 1

CI 255

*(reserved)*

Notes

ONBUS is transmitted by a bus member right after joining the bus. It is afterwards transmitted again automatically by an active bus member at least every 300 seconds, but not more often than once every 60 seconds.

OFFBUS is transmitted when a device is about to shut down and be no longer reachable through the bus without certain activity that cannot be initiated remotely through the bus itself. It is, therefore and due to protocol specification, not transmitted when a device goes into the “passive” or “off” state after a CAN protocol error.

SLEEPING is transmitted before a device is entering a sleep mode where it might consume less energy than in normal mode. To become active, this device might require an ONBUS remote frame request to wake it up. After waking up, the device is supposed to send an ONBUS message.

E_COUNTER is only transmitted on request by another bus device, usually the HA-bus control station.

E_OFLOW is only transmitted on request by another bus device, usually the HA-bus control station.

E_ERRORS is transmitted by a device after it has cleared one of the reported CAN error states. It can also be requested by another bus device, usually the HA-bus control station.

S_IOSTATE transmits state information for digital inputs and outputs as defined for MTID 06 OUTDIG, for multi-valued inputs and outputs as defined for MTID 07 OUTVAL.

D_RANGE values are, as all data transmitted in single bytes, ranging 0..255.

I_WTIME is transmitted by time sources (such as the control computer) about every 20 seconds.

The control station is allowed to issue a remote transmit request for MTID 12 directed to any bus device address. The bus device, if present, is expected to respond to this by sending one or more MTID 13 datagrams. The addressed device is required to respond with the respective MTID 13 message(s) as soon as possible, with a maximum delay of 300 ms. If no actual response to the request is possible within that timeframe, the bus device is required to send a CI 001 ONBUS message within that timeframe, and send the response when it is ready. The number of MTID 13 response datagrams and additionally transmitted CIs is device specific, and documented in the respective part description.

*MTID 14, BULKDATA: Bulk data transfer protocols*

This message type is used for transmission of data to be assembled on the receiving side. Currently, this MTID is used for the transfer of firmware update files from the HA control computer to connected devices and for the automatic CAN address assignment protocol (ACAA) as described in HA-I01 (page 87).

The first data byte of any BULKDATA transmission identifies the sub-protocol being used. The following identifiers have been assigned:

| DATA[0] | Protocol | Documentation |
|---|---|---|
| 10 | device firmware update | HA-I12 (p. 133) |
| 70 | ACAA | HA-I01 (p. 87 |
| 71 | ACAA | HA-I01 (p. 87 |
| 72 | ACAA | HA-I01 (p. 87 |
| 73 | ACAA | HA-I01 (p. 87 |

For different types of data, different sub-identifiers and sub-protocols may be defined. The firmware update mechanism as explained in HA-I12 (page 133) may servce as an example for future extensions.

### 16.1.3. Device identifier

The following device identifiers (DIDs) are reserved for special purposes or future use. They must not be assigned to bus devices.

| DID | Description |
|---|---|
| 000 | USB/CAN interface HA-B02 |
| 121 | Automatic CAN Address Assignment (ACAA) HA-I01 |
| 122 | (reserved) |
| 123 | (reserved) |
| 124 | (reserved) |
| 125 | (reserved) |
| 126 | (reserved) |
| 127 | (reserved) |

# 17. HA-I03, Device Type Identifiers (DTIDs)

| *HA-I03.01*: Production |
|---|

This interface definition provides the registry of assigned device type identifiers.

## 17.1. HA-I03.01

Revision HA-I03.01 is updated with new entries under the same revision number.

| DTID | Description |
|---|---|
| 1 | HA-B01 multi-channel digital input |
| 2 | HA-B02 dual-CAN/USB control PC interface |
| 3 | *TBD* multi-channel digital switch controller |
| 4 | *TBD* multi-channel dimmer output controller |
| 5 | `iod_can_emulator` dummy device |
| 6 | HA-B06 modular CAN-I/O bus module |
| 7 | HA-B07 n-way controller interface (on HA-B06) |
| 8 | HA-B08 16-way input interface (on HA-B06) |
| 9 | HA-B04 6-input digital input module, in-wall |

Updates to this table need to be also included in software components:

- In `ha-p03/ha_iod`, `can_dtidtab` in file `can_dtid.c`
- In `ha-p03/ha_common`, `dtid_input` and `dtid_output` in file `deviceclass.py`

# 18. HA-I04, Communication of ha_iod to TCP client software

| *HA-I04.01*: Production |
|---|

This protocol definition describes the protocol and message format for data exchange between the USB I/O dispatcher daemon `ha_iod` and its TCP clients.

## 18.1. HA-I01.01

### 18.1.1. Messsage format

Messages are transmitted as new-line terminated lines of ASCII characters.

The general format of a message is defined as:

```
CC data\n

CC   command code
data as defined below
\n   terminating new–line character
```

The terminating new-line character (ASCII code 10) is not shown in the further protocol details, but must be present in the actual datagrams.

### 18.1.2. Message types

#### CD - Send CAN message

```
Client: CD X NNNN MM L DDD DDD DDD DDD DDD DDD DDD DDD
ha_iod: CD:ok
        CD:INVALID
        CD:USBGONE
        CD:CAN_ERR [Details of the error returned by the CAN attachment
          controller]
```

**X** either M requesting a standard CAN transmission, or R requesting a remote transmit request to be sent.

**NNNN** the four-digit CAN device number, including the bus ID encoded in the upper five bit of this 16-bit value.

**MM** the numeric message type ID

**L** one numeric character, indicating the number of data bytes in this transmission

**DDD** the data bytes to be sent, if any, in their decimal string representation.

#### CL - CAN device list

```
Client: CL
ha_iod: CL:start
        CL:nnnn:lastseen:ident string
        CL:end
```

`ha_iod` keeps a list of CAN devices attached to the bus, which can be queried with this command.

#### CR - Received CAN message

```
ha_iod: CR X NNNN MM L DDD DDD DDD DDD DDD DDD DDD DDD
        (no response from client expected)
```

**X** either N indicating a standard CAN transmission, or E requesting a remote transmit request that was received.

**NNNN** the four-digit CAN device number, including the bus ID encoded in the upper five bit of this 16-bit value.

**MM** the numeric message type ID.

**L** one numeric character, indicating the number of data bytes in this transmission.

**DDD** the data bytes that were received, if any, in their decimal string representation.

#### PM - Measure CAN channel current consumption

```
Client: PM B
ha_iod: PM:B:AAAA
        PM:ERR:BUSY
        PM:ERR:INVALID
        PM:ERR:UNSUP
```

**B** the bus number to measure.

**AAAA** the four-digit current measurement as raw value read from the bus controller's analog-to-digital converter (ADC). Calculation of the actual current in Ampere from this value is hardware-dependent and described in the respective hardware documentation.

#### PS - CAN controller power state report

```
Client: PS
ha_iod: PS:NN:AA:BB:...
        PS:INVALID
```

**NN** the number of bus channel states to follow.

**AA, BB, and so on** the bus channel states as measured: `00` means the bus power is turned off, `01` means it is turned on.

#### PW - CAN controller power switching

```
Client: PW B O
ha_iod: PW:OK:NNN
        PW:HWERR
        PW:INVALID
        PW:USBGONE
```

**B** the bus number. Currently, `0` and `1` are supported.

**O** the power state to switch to, either `0` for “off”, or `1` for “on”.

**NNN** the time it took the respective power switching relay to throw, in miliseconds.

#### RS - Reset CAN controller

```
Client: RS X
ha_iod: RS:OK
        RS:INVALID
        RS:USBGONE
```

**X** the CAN bus number requiring a controller reset (currently, `0` and `1` are supported).

#### Error messages

```
Client: [invalid command]
ha_iod: ER:001 Invalid command.
```

# 19. HA-I05, Communication USB/CAN interface to ha_iod

| *HA-I05.01*: Production |
|---|

## 19.1. HA-I05.01

This protocol definition describes the communication protocol between the USB/-CAN converter HA-B02 and the control PC computer program `ha_iod`.

### 19.1.1. Transport layer

1. The messages between `ha_iod` and HA-B02 are exchanged over a USB-serial link, running at 115200 bps, 8 data bits, no parity, 1 stop bit (”8N1”).
2. No method for flow-control is used.

### 19.1.2. Data format

Data is transmitted using ASCII characters ranging from 33-160.

1. Control characters are to be interpreted as such, and are always the first character of a transmitted datagram.
2. Datagrams are terminated by the line-feed character (ASCII code 10).
3. Numerical data is transmitted as two byte characters. Their ASCII codes are calculated on encoding by adding the value of `!` (33) to the upper and lower four bit of the byte to be transmitted. The first byte transmitted represents the upper four bit of the data byte, the second byte transmitted therefore represents the lower four bit of the data byte.
4. The elements of one datagram, that is the control character and each two-byte ASCII encoded character, are each separated by space characters (ASCII value 32).

### 19.1.3. Datagrams

Data needs to be transmitted in different directions between different parts of the stack. In the following subsection, “CAN” means the CAN bus and its attached devices, so a message “to CAN” is a packet of data that needs to be sent out through the CAN bus. On the other hand, a message “to HA-B02” is a message that needs to be interpreted by the HA-B02 controller itself, and does not have to be forwarded to CAN (but may trigger other actions on the CAN bus).

In the following format specifications, capital letters are placeholders for data explained in the respective description, small letters are control characters and transmitted as-is. Numerical values given here are transmitted as their ASCII representation as described in “Data format” above.

#### ha_iod to CAN

```
m HH LL NN DD DD DD DD DD DD DD DD
r HH LL NN DD DD DD DD DD DD DD DD
```

If “m” is used as control character, a standard CAN message is to be sent.

If “r” is used as control character, the remote transmit request (RTR) bit is to be set on the outgoing message. As defined by the CAN standard, no data bytes are allowed in the message in this case.

H Upper 8 bit of 16 bit CANID (including bus identifier)

L Lower 8 bit of 16 bit CANID

N Amount of data bytes to transmit (0..8)

D 8 bytes of data in two-digit representation. Unused bytes are padded with 0 characters.

#### CAN to ha_iod

```
n HH LL NN DD DD DD DD DD DD DD DD
e HH LL NN DD DD DD DD DD DD DD DD
```

If “n” is used as control character, a standard CAN message has been received.

If “e” is used as control character, the remote transmit request (RTR) bit was set on the incoming message.

H Upper 8 bit of 16 bit CANID (including bus identifier)

L Lower 8 bit of 16 bit CANID

N Amount of data bytes that follow (0..8)

D 8 bytes of data in two-digit representation. Unused bytes are padded with 0 characters.

### 19.1.4. Control messages

#### CAN controller reset

```
a
b
```

This command can be sent from `ha_iod` to CAN, causing it to reset and reconfigure the CAN controller for either bus A or B (bus address 0 or 1, respectively).

#### CAN bus current measurement

```
O B
```

This command is sent from `ha_iod` to read the current current consumption one CAN bus line. `B` is the bus number to read, 0 representing bus A, 1 representing bus B, and so on.

HA-B02 responds with one line containing the requested measurement:

```
O:01:0130
```

The first field confirms the channel that was requested and read, the second field contains a four-digit value representing the measured current consumption in milliamperes.

The measurement information can also be sent by the USB/CAN interface without a preceding request from the control station computer, for example if certain current thresholds are exceeded.

If the USB/CAN interface hardware does not support current measurement on the bus lines, as for example HA-B02.01 does, the device responds with the line

```
O:ERR:UNSUP
```

If a measurement is already in progress on the interface hardware, and a new request cannot be served at the time it arrived, the response is

```
O:ERR:BUSY
```

#### CAN bus power sensing

```
o
```

This command is sent from `ha_iod` to read the current power state of the CAN bus lines. HA-B02 responds with one line indicating the power states of all attached CAN buses:

```
o:02:01:01
\end{listlisting}

The first field contains the number of power state information fields to
come, the remaining colon-separated fiels indicate the power state:
\cnst{00} means powered off, \cnst{01} means powered on.

\subsubsection{CAN bus power switching}

\begin{lstlisting}
p B P
```

B is the bus number, encoded as described for numerical values above.

P is the power state to switch to, either 0 or 1, also encoded as numerical values as above.

The CAN controller will respond with one of the following messages:

```
p:OK:NNN
```

with NNN being the number of milliseconds it took the relay to throw (tested value for freshly bought relay is 5-6 ms)

```
p:ERR:HW
```

in case of relay timeout or other hardware errors

```
p:ERR:INV
```

in case of an invalid P for this command

**HA-B02 identification**

```
i
```

The control character “i” can be sent from `ha_iod` to HA-B02 to request hardware and firmware version information. The response string will be prefixed with the character “i” as well.

For reference, the current response to this command is the following:

```
inz@thiemo.net:HA-B02.01:HA-P04.01:dev
\bein{lstlisting}

The response string is terminated with the two-character sequence
cursor-return, line-feed.

\subsubsection{Reserved}

The following control characters are reserved for testing purposes and
should not be used in regular operations.

\begin{lstlisting}
t
```

Sends a test CAN message in development build. Later to be used for entering a USB/CAN interface test mode.

# 20. HA-I06, Internal CAN connector specification

| *HA-I06.01*: Production |
|---|

## 20.1. HA-I06.01

This interface description defines the internal CAN bus connector type and pin assignment. Internal bus connectors are used within an assembly to connect individual devices. External connectors are specified in another interface description, HA-I07 (page 115).

### 20.1.1. Physical connector

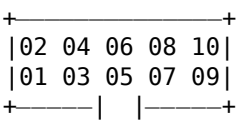
A rectangular pin header connector, $2 \times 5$ contacts, is used. Pin spacing is 2.54 mm.

A mechanically coded connector is used, with pin numbering according to the included drawing.

Circuit boards are equipped with pin headers, interconnect cables therefore use receptacle connectors.

### 20.1.2. Pin assignment

PCB pin header, from top side *XXX* Make real drawing XXX




The pin numbers given in the mechanical drawing are assigned to signals as follows. Note that this assignment does not reflect the `CANopen` recommendation for 10-pin interconnect headers.

Pin assignments:

| | |
|---|---|
| 01 | CANH (0-5V DC) |
| 02 | CANL (0-5V DC) |
| 03 | GND |
| 04 | +12V (max. bus total 1 A) |
| 05 | RSVD_5 (reserved) |
| 06 | GND |
| 07 | GND |
| 08 | +12V (max. bus total 1 A) |
| 09 | NC (not connected) |
| 10 | NC (not connected) |

# 21. HA-I07, External CAN connector specification

| |
|---|
| *HA-I07.02*: Production<br>*HA-I07.01*: Obsolete |

This interface description defines the external CAN bus connector type and pin assignment. External bus connectors are used to connect seperated assemblies that may be more or less physically separated. Internal connections are defined in HA-I06 (page 113).

## 21.1. HA-I07.02

### 21.1.1. Physical connector

A spring-loaded or screw-fastened 10-way plastic insulated terminal block is used. Electrical rating is at least 60 VDC, 2 A (continuous current). Viewing the block's top from the side where the bus cable wiring is connected, pin numbers are counted from left to right, starting with 1.

An example for the spring-loaded variant is Phoenix Contact, "COMBICON PTSA 0,5/10-2,5-F" (product number 1985027).

The bus cable's wires are directly inserted into the terminal block and secured as required by the respective terminal block type.

### 21.1.2. Pin assignment

Note that this assignment does not reflect the CANopen recommendation for 10-pin interconnect headers. It is, however, identical with the HA-I06 specification for internal CAN bus connectors, and pin-compatible to the previous HA-I07.01 revision of this specification for external CAN bus connectors, though it uses a different physical connector.

Pin assignments:

| | |
|---|---|
| 01 | CANH (0-5 VDC) |
| 02 | CANL (0-5 VDC) |
| 03 | GND |
| 04 | +12V (max. bus total: 1 A) |
| 05 | RSVD_5 (reserved) |
| 06 | GND |
| 07 | GND |
| 08 | +12V (max. bus total: 1 A) |
| 09 | NC (not connected) |
| 10 | NC (not connected) |

## 21.2. HA-I07.01

Obsolete

This interface description defines the external CAN bus connector type and pin assignment. External bus connectors are used to connect seperated assemblies that may be more or less physically separated. Internal connections are defined in HA-I06 (page 113).

### 21.2.1. Physical connector

A FCC Registered Jack 8P8C “RJ45” connector is used. Pins are numbered according to the sketch below. Connector shielding is not required. If metal shields are used on connectors, care must be taken not to locally ground these shields to prevent ground loops over greater distances. In this case, the bus controller serves as the single grounding point for the connector shielding. The bus connector and cabling includes one line to connect local shielding to the central shield grounding point.

Regular bus devices have two bus connectors to minimize stub lines off the bus cabling. The bus controller therefore has a single output connector per bus, as this already includes one termination resistor. The remote end passive bus terminator also has a single connector, because the bus cable ends there.

Device connectors are implemented as receptacles (“female” contacts), cables are equipped with the matching plugs (“male” contacts) on both ends.

### 21.2.2. Pin assignment

| Pin | Signal | Remarks |
|---|---|---|
| 1 | CANH | |
| 2 | CANL | |
| 3 | GND | |
| 4 | +12V | This line is ”reserved” in CANopen. |
| 5 | (unused) | |
| 6 | Shield (optional) | |
| 7 | GND | |
| 8 | +12V | |

The +12 V line is split over two contacts to allow for less current loading on the bus cabling. The +12 V and GND lines should be connected within devices. This assigment follows the CANopen recommendation for “RJ45” type connectors as far as signal lines are concerned, but uses pin 4 as additional supply line, where CANopen lists this pin as “reserved”. Care should be taken when adding third party CAN bus devices to an HA-CAN bus.

# 22. HA-I08, Data exchange protocol between ha_control and its user-side components,

| *HA-I08.01*: Development |
|---|

This interface description defines the XML-RPC interface served by `ha_control` and used by other HA software components.

# 23. HA-I09, Conventions for variable identifiers in HA software

| *HA-I09.01*: Development |
|---|

To improve source code readability and ease the understanding of code when encountering it at later times, identifier names for certain uses are standardized in this document. Old code shall be changed to match this standard, and new code must adher to the definitions and rules made herein.

# 24. HA-I10, Interface for CAN bus device low-level configuration

| |
|---|
| *HA-I10.02*: Development<br>*HA-I10.01*: Development, obsolete |

CAN bus devices contain a low-level configuration and debugging interface through which certain settings can be made that must be possible without a CAN bus connection. The most important and most frequently used setting is the initial assignment of the CAN device identifier (DID) before a device becomes active on the bus. The physical interface and the protocol on the bus are defined in this document.

## 24.1. HA-I10.02

### 24.1.1. Changes from previous revision

This revision changes from the I²C connector and protocol to a UART-based communication protocol.

### 24.1.2. Physical interface

Communication takes place over a UART interface presented on a 4-pin, 2.54 mm pitch, plain header connector. Pin 1 is marked on the circuit board. The UART signals available on the connector operate on 5 V levels, the supply voltage provided is also +5 V.

Devices plugging into the UART connector must not consume more than 50 mA from the `+5V` supply pin. The `UART_TXD_5V` pin must not be loaded with more than *XXX* mA to prevent excessive voltage drops through the included overcurrent protection resistor. XXX

Pin assignments:

| | |
|---|---|
| 1 | GND |
| 2 | UART_TXD_5V |
| 3 | +5V |
| 4 | UART_RXD_5V |

Note that “TXD” and “RXD” are viewed from the CAN bus device’s MCU. On the board connector, on UART_TXD_5V, data from the MCU is presented to a connected configuration and debug device. UART_RXD_5V acts as the data input to the bus device MCU.

### 24.1.3. Protocol

With the UART interface being a point-to-point connection, no addressing scheme is required for this interface.

Communication parameters are 9600 bit per second, 8 data bit, no parity, 1 stop bit (“9600/8N1”). No flow-control mechanism is applied.

The bus device accepts text commands terminated with a new-line character (ASCII code 10) as listed below. The commands given here are case-sensitive. On successful command completion, the device responds with the string “Ok.” and the new-line character. In other cases, an error message is returned.
The device can also output diagnostic messages in text format, again terminated with new-line characters.

#### Commands

| | |
|---|---|
| cfg wr | Write the current configuration data to EEPROM. If this command is not issued after setting other config options, these changes will be lost after a device reset. |
| did get | Print the currently configured CAN DID. |
| did set \vrbl {NNN} | Configure the CAN device identifier (DID). NNN must be given in three-digit format, and must be within the range for CAN DIDs as specified in the CAN higher layer protocol HA-I02 on page 89. Currently (as of HA-I02.01), this range is 001 to 127. |
| module ident | Print the identification string and firmware index for the attached I/O module as determined after reset. |

# 24.2. HA-I10.01

Obsolete

## 24.2.1. Physical interface

The I²C connector is a 5-pin, 2.54 mm pitch, plain header connector. Pin 1 is marked on the circuit board. The I²C bus lines available on the connector operate on 5 V logic levels, the supply voltage provided is also +5 V.

Pin assignment is defined as follows:

Pin assignments:

| | |
|---|---|
| 1 | GND |
| 2 | I2C_INT |
| 3 | +5V |
| 4 | SDA |
| 5 | SCL |

*I²C interrupt*

With the CAN bus device acting as an I²C master, the attached configuration device needs a way to request attention from the master. For this purpose, an interrupt signal has been added to the I²C signals that can, for example, be connected to one of the bus master's external interrupt lines. The net name has therefore been assigned as I2C_INT. Note that this signal is defined as active-positive, that is, when idle, 0 V are present on the line, when an interrupt is to be triggered, the positive-going edge to 5 V should be detected. The I²C slave device leaves I2C_INT at a high level as long as communication with the master is desired.

The I2C_INT signal line available on the I²C bus connector is kept in a stable state by a 10 k resistor to GND.

*Data lines*

The SDA and SCL data lines are short-circuit protected by 180 R series resistors. Current consumption from the data lines should therefore be minimized, which usually is not a problem when using CMOS devices on the interface.

## 24.2.2. Protocol

The I²C configuration and debug device is addressed as a slave device at I²C address 0x63.

When powering up a fresh and unconfigured CAN bus device for the first time, the CAN bus device sends a I2C_CFG_CAN_DEVID (0x01) to the slave, and expects

to receive the numerical CAN device identifier as one byte. If no response is received from the bus, this request is periodically retried. Until a CAN address has been configured, the device will not engage in CAN bus communications.

# 25. HA-I11, Modular CAN-I/O interface definition

| *HA-I11.01*: Production |
|---|

This definition describes the interface between the Modular CAN-I/O Bus Logic Board (BLB) and the Controller/Sensor peripheral boards.

## 25.1. HA-I11.01

### 25.1.1. Board layout and connector positions

The board consists of 1.5mm FR4 board or material of similar specifications.

Width is 160mm, height 100mm.

There are four mounting holes of diameter 3.2mm to match the M3 mounting screws used. These are located at the following coordinates, with the origin in the lower left corner of the board.

| x (mm) | y (mm) |
|---|---|
| 6.6 | 6.6 |
| 6.6 | 93.4 |
| 153.4 | 93.4 |
| 153.4 | 6.6 |

The module interconnect is a $2 \times 20$-pin, 2.54 mm pin-spacing, orientation-coded pin header. Its center is located at (55.0 / 90.0) milimeters, pin 1 is on the left hand side, lower row. The orientation coding notch on the connector casing is pointing downwards, that is on the “pin 1” row. Pin assignment and interface specifications are defined below.

If the board layout requires so, this connector may be shifted in Y direction, but should be placed no lower than at $Y = 55$ mm.

CAN bus connectors on the BLB are located on the right hand side of the board. Connector X placement is so that the connector housing ends in line with the board boundary. Vertically, the connectors are centered at $Y = 35.0$ mm (CAN1) and $Y = 70.0$ mm (CAN2). If internal CAN bus connectors are mounted, pin 1 is to

be placed in the lower right corner, when viewed from the top board side. Internal CAN bus connectors may leave up to 3 mm space between the board boundary and the connector housing.

If present, the I²C configuration connector is a 5-pin, 2.54 mm, plain pin header, its center located at (100.0 / 90.0) milimeters. Pin 1 is on the left hand side. Pin assignment is defined in HA-I10, page 123.

If present, a UART connector can be present as 4-pin, 2.54 mm, plain pin header, the center located at (100.0 / 80.0) milimeters. Pin 1 is on the left hand side. Pin assigment is currently undefined.



### 25.1.2. Physical interface

The Modular CAN-I/O interface connects the two modules (BLB and peripheral board) through a 2 × 20-pin header as specified above. The PCBs are equipped with pin headers “male”, while the connecting flat ribbon cable carries the matching receptacles “female”.

The pin assignment is as follows:

| Pin | Lvl | Description | Pin | Lvl | Description |
|---|---|---|---|---|---|
| 1 | 3V3 | OUT_1 | 2 | 3V3 | IN_1 |
| 3 | 3V3 | OUT_2 | 4 | 3V3 | IN_2 |
| 5 | 3V3 | OUT_3 | 6 | 3V3 | IN_3 |
| 7 | 3V3 | OUT_4 | 8 | 3V3 | IN_4 |
| 9 | 3V3 | OUT_5 | 10 | 3V3 | IN_5 |
| 11 | 3V3 | OUT_6 | 12 | 3V3 | IN_6 |
| 13 | 3V3 | OUT_7 | 14 | 3V3 | IN_7 |
| 15 | 3V3 | OUT_8 | 16 | 3V3 | IN_8 |
| 17 | 3V3 | OUT_9 | 18 | 3V3 | IN_9 |
| 19 | 3V3 | OUT_10 | 20 | 3V3 | IN_10 |
| 21 | 3V3 | OUT_11 | 22 | 3V3 | IN_11 |
| 23 | 3V3 | OUT_12 | 24 | 3V3 | IN_12 |
| 25 | 3V3 | OUT_13 | 26 | 3V3 | IN_13 |
| 27 | 3V3 | OUT_14 | 28 | 3V3 | IN_14 |
| 29 | 3V3 | OUT_15 | 30 | 3V3 | IN_15 |
| 31 | 3V3 | OUT_16 | 32 | 3V3 | IN_16 |
| 33 | GND | GND | 34 | GND | GND |
| 35 | 5V | I2C_SCL (4.7 k pull-up on BLB) | 36 | 5V | I2C_INT (10 k pull-down on BLB) (*) |
| 37 | 5V | I2C_SDA (4.7k pull-up on BLB) | 38 | | (reserved, unconnected) |
| 39 | 5V | +5V supply | 40 | 3V3 | +3V3 supply |

(*) The BLB always acts as the I2C bus master. Through the I2C_INT line, peripheral devices can signal the need to be queried for information by the master. The fact that only one interrupt request line is shared by multiple bus devices is considered in the higher levels of this interface specification.

The I2C_* lines are short-circuit protected on the BLB by series connection through 180 R resistors. I²C bus pull-up resistors (4.7 k) are incorporated in the BLB as well. The I2C_INT line is pulled down through 10 k on the BLB.

A peripheral module may use the +5 V and +3.3 V supply lines as power supply for its own needs within specified limits. Note that between the supply and logic voltages on the peripheral module connector and the infrastructure controlled by the bus device, an additional layer of protection is required by the HomeAutomation design specifications, if the infrastructure device has electrically conductive connections into an enclosure containing other parts that carry 230 V (or 110 V, if applicable) utility power.

The power consumption on the module connector supply lines must not exceed the limits placed by the power regulator circuitry on the BLB. The power budget

on the +3.3 V supply is equivalent to 0.2 A, for the +5 V supply it is 0.7 A. Recommended maximum supply line loading is 25 mA for the +3.3 V supply and 250 mA for the +5 V supply. Note, however, that the total CAN bus power consumption must not exceed 1 A at 12 VDC per bus line, so power efficiency should be generally considered in the design of peripheral modules.

As an example, when opto-isolated switch inputs are used and periodically polled for changes, it is unnecessary to keep the optocouplers powered at all times. The 16-way input controller designed and described herein (HA-B08) assignes four input channel groups, each containing four channels (and therefore, one quad optocoupler), and manages power supply to the couplers' LEDs through module-specific I²C bus commands. Thus, when polling the switches every 10 msec, and a poll taking about 0.5 msec, overall power consumption is decreased by a factor of nearly 20, and peak consumption is reduced to 25

### 25.1.3. Protocol

The `OUT_n` and `IN_n` lines are used to control up to 16 input or output devices. `OUT_n` lines are outputs driven by the BLB, while `IN_n` lines are inputs to be read by the BLB and driven by the peripheral board. The specific use of these signals depends on the peripheral board's functionality and is documented in the respective module part description.

The `I2C_*` lines are used for control communications between BLB and peripheral module. When the bus device is powered up, the BLB first queries the peripheral board for its hardware identification through the I²C bus, and then either operates it in the way required for the module that has been detected, or reports a configuration error to the operator if it is not programmed to handle the respective peripheral module.

To identify the peripheral board, the BLB sends a packet with an `MODCOMM_IDENTIFY (0x01)` command to it at I²C bus address `0x4a`, and then reads four data bytes:

| Byte | Value |
|---|---|
| 00 | Module ID (see below) |
| 01 | Module firmware ID |
| 02 | Input channel count |
| 03 | Output channel count |

The module ID is assigned according to the DTID registry in HA-I03 (page 101), as the BLB is supposed to assume its DTID for transactions on the CAN bus.

If the BLB I²C bus master requires an I²C bus address, `0x49` is reserved for this purpose. It must not be used by devices integrated in the peripheral module.

The following I²C bus addresses are assigned to fixed devices, and must not be used by any device integrated in the peripheral module:

| Address | Device | Specification |
|---|---|---|
| 0x49 | BLB MCU as I²C device | HA-I11 |
| 0x4a | Peripheral board MCU | HA-I11 |
| 0x52 | BLB config EEPROM | HA-I11 |
| 0x63 | Low-level configuration and debug connector | HA-I10 |

Any other protocol messages and functionality of I²C devices on the bus is specific to the peripheral module and documented in its part description.

# 26. HA-I12, CAN firmware update protocol

| *HA-I12.01*: Development |
|---|

## 26.1. Description

Firmware running on CAN-attached devices can be updated through the bus. The protocol defined here allows the new firmware data to be transferred during normal operations of the devices and bus infrastructure, keeping the outage required for actually programming the devices as short as possible.

The Modular CAN-I/O Bus Logic Boards carry an amount of flash memory capable of storing two device firmware images as well as additional local configuration information. During normal operation, the control computer (or any other sufficiently equipped bus station) can transfer a firmware image to the devices to be programmed in small packets over the CAN bus connection.

The devices write these packets to their on-board flash memory while otherwise continue to function as regular bus devices. Once the complete new firmware image has been transferred, they can react to a special command from the control station, switching them into the actual programming mode. Upon entering this mode, normal operations are not available on the device being programmed. The bus device is then reading from the local flash memory the new firmware image, and programs it into its actual program memory section. After having programmed the complete new firmware, the device resets and tries to start the new firwmare just loaded.

If this start of new code is successful, the new image on the local flash memory is marked as “good”, and the next firmware update file will be written to the second portion of storage space reserved for firmware update images. If the start of new code fails and leads to a device reset (by local watchdog action, local power reset, or bus power reset), the new firmware image is marked as “bad”, the other present last known good image is being programmed into the controller’s program memory, and started.

## 26.2. Protocol

The transmission of new firmware images uses the `BULKDATA` transfer channel as defined in the HA-I02 CAN higher layer protocol specification. Subchannel identifier `10` has been assigned for the purposes of firmware update transmissions.

# Part V.

# Registries and notes

# 27. Part number reference

*Note:* The status information provided here relates to the part as a whole, not to a specific revision of the part.

## 27.1. HA-A... Assemblies

| Identifier | Description | Status |
|---|---|---|
| HA-A01 | Modular HA-CAN 16-channel binary output controller | production |
| HA-A02 | Modular HA-CAN 16-channel binary input controller | production |
| HA-A03 | 6-button in-wall switch input assembly | development |

## 27.2. HA-B...: Circuit boards

| Identifier | Description | Status |
|---|---|---|
| HA-B01 | 16-way digital output controller, CAN attached | discontinued |
| HA-B02 | USB dual-CAN bus interface | production |
| HA-B03 | uC/CAN interface module | discontinued |
| HA-B04 | 6-input digital input module, in-wall, CAN-attached | development |
| HA-B05 | 8-way relay controller output module | discontinued |
| HA-B06 | Modular CAN-I/O module, bus logic board | production |
| HA-B07 | Modular CAN-I/O module, 16-way output module | production |
| HA-B08 | Modular CAN-I/O module, 16-way input module | production |
| HA-B09 | CAN Internal to External connector adapter | development |
| HA-B10 | Modular CAN-I/O module, 2-way output test module | development |
| HA-B11 | XBee-CAN interface | development |

## 27.3. HA-D...: CAD drawings, assembly plans, manuals

| Identifier | Description | Status |
|---|---|---|
| HA-D01 | Checklist for porting HA-P01 to new bus devices | moved to doc |
| HA-D02 | HomeAutomation documentation | development |
| HA-D03 | 3D models of PCB mounting hardware | development |
| HA-D04 | CAD model of 6-button in-wall switch input assembly HA-A03 | development |



## 27.4. HA-F...: Mechanical parts

| Identifier | Description | Status |
|---|---|---|
| HA-F01 | PCB holder, screw mount, sliding | development |
| HA-F02 | Case for wireless lighting and temperature sensor | development |
| HA-F03 | Case for I/O module installation units | development |
| HA-F04 | Case for 6-button in-wall switch input assembly HA-A03 | development |

## 27.5. HA-I...: Interface descriptions, protocols



| Identifier | Description | Status |
|---|---|---|
| HA-I01 | Protocol definition for automatic CAN address assignment (ACAA) | development |
| HA-I02 | Home-CAN protocol description | production |
| HA-I03 | Bus device type identifiers | production |
| HA-I04 | Data exchange protocol between `ha_iod` and TCP clients | production |
| HA-I05 | Data exchange protocol between HA-B02 USB device and `ha_iod` | production |
| HA-I06 | Definition of internal CAN bus connector type and pin assignments | production |
| HA-I07 | External CAN connector specification | production |
| HA-I08 | Data exchange protocol between `ha_control` and its user-side components | development |
| HA-I09 | Conventions for variable identifiers in HA software | development |
| HA-I10 | I²C connector for CAN bus device low-level configuration | development |
| HA-I11 | Modular CAN-I/O interface definition | development |
| HA-I12 | CAN firmware update protocol | development |

## 27.6. HA-P...: Programs

| Identifier | Description | Status |
|---|---|---|
| HA-P01 | Firmware template for CAN-attached modules | production |
| HA-P02 | Control program with graphical user interface | development |
| HA-P03 | Low-level host side (control station) software bundle | production |
| HA-P04 | Firmware for HA-B02 (USB dual-CAN bus interface) | production |
| HA-P05 | Firwmare for HA-B06 (modular CAN-I/O bus logic board) | prototype |
| HA-P06 | Firmware for HA-B07 (CAN-I/O 16-way controller module) | development |
| HA-P07 | Firmware for HA-B08 (CAN-I/O 16-way input module) | development |
| HA-P08 | Firmware for HA-B04 (6-input digital input module, in-wall, CAN-attached) | development |

# 28. Known issues and “to do” items

This section serves as a reminder for issues found during system and component testing. It does not comprise a changelog, as the history of issues in hardware and software found and fixed is kept in the HA revision control system.

## 28.1. Hardware

### 28.1.1. HA-B02

Obsolete

**Include current shunt resistors for both CAN channels, connected to MCU’s ADCs**

- 12V, 1A max
- ADCs have 10bit resolution, 2.56V internal reference -> 2.5mV/bit

  -> max current is then 1.27A

  ADC accuracy is +/- 2 LSB
- 0R5 -> 0.5V drop@1A, multiply by 4 in op-amp voltage buffer

  R(cs) power rating >= 1W (200
- 0R1 -> 0.1V drop@1A, multiply by 20 in op-amp voltage buffer

  R(cs) power rating >= 0.2W (200
- The 0R5 variant might be preferable for higher noise tolerance
- check AVcc decoupling of internal voltage reference in Atmega32 datasheet
- ADC0(PA0) and ADC1(PA1) are unused in current design
- 0.5R w/ 1

  2LSB is 6.25mV, 6.25mV/4=1.56mV -> want 0.1
- So, current shunt resistor should be: 0.5R, >=1W, <=0.1

## 28.2. Software

### 28.2.1. USB/CAN interface stack

#### Transmissions CAN->USB lock up while USB->CAN still works

- It is unknown when and where exactly this happens...
- The CANH and CANL lines show transmissions in both directions.
- The HA-B02 MCU toggles the `!USB_CTS` signal when it hangs, so at least the interrupt routine does work. It also asserts it for the `ST_FIRED_500MSEC` flag every 500 msec, so the main loop must be running as well. However, no activity is seen on the `USB_RXD` line (where the MCU should send the outgoing data).
- The `!CANA_INT` line is permanently active (low).
- There is no activity on the `SPI_MISO` and `SPI_MOSI` lines.

### 28.2.2. HA-P03

Obsolete

#### hashell has issues with bus 1

- When parsing the ”rly rq all” output enumeration response, it is ignored due to devid mismatch (because the bus id is ignored).
- Likewise, in output strings, only the devid is printed, but the bus id is not calculated into it.

### 28.2.3. HA-P05

#### Scoping shows missing CAN ACKs on transissions from the USB/CAN interface to the BLB.

- Back direction is shown as ACKed.

## 28.3. Notes

- Request/response from USB/CAN to BLB: Start of CAN request to CAN interrupt on USB/CAN deasserted is 1.9 ms (three data bytes). (Before buffer copyout mechanism was introduced!)
- SOF of three data byte CAN datagram until interrupt is deasserted takes 830 us. (Before buffer copyout mechanism was introduced!)

- Three data byte CAN datagram is transmitted in 560 us. (Before buffer copy-out mechanism was introduced!)

# 29. Glossary

**ACAA** Automatic CAN Address Assignment
**ADC** Analog-to-Digital Converter
**BLB** Modular CAN-I/O module, Bus Logic Board
**CAN** Controller Area Network
**CI** Command Identifier
**COTS** Commercial off the shelf, i.e. commercially available in the free market
**DID** Device Identifier
**DTID** Device Type Identifier
**HACS** Home Automation Control System
**LSB** Least Significant Bit
**MCU** Microcontroller Unit
**MTID** Message Type Identifier
**PCB** Printed Circuit Board

# Index

Page numbers set in **b**old typeface are references to the definition of the respective terms. Pages where the term is mentioned in a context the author considers useful are set in normal typeface.