**Communication Procedure OBC to PSU:**

1. Connections
   1. Atmega128A (MC)

To program the main controller (ATmega128A), it is necessary to connect the Atmel Ice Debugger with the programming pins which are located on the Y+ (former D) side connector. Therefore, plug in the flat cable on the “AVR” pins at the Atmel Ice Debugger → **Figure 2** and connect the prepared plug on the pins 31 to 40 on the side connectors Y+/D → **Figure 1**.

Ein Bild, das Text, Elektronik, Schaltkreis enthält.

Automatisch generierte Beschreibung

Figure 1-Plug connection to Atmega128A @ PSU

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Figure 2-Atmel Ice Debugger Connection

* 1. Power Supply PSU

To supply every Controller and I²C slave with the necessary power just connect the “+” output of the power supply with the side connector pin “PV1” or with “PV2”.

PV1 Pin → Pin 3 & 4 on each side connector → **Figure 3**

PV2 Pin → Pin 11 & 12 on each side connector → **Figure 3**

The “-“ output has to be connected to the pin “GND” on the side connectors.

GND → Pin 1, 2, 13, 14, 39 & 40 on each side connector. → **Figure 3**

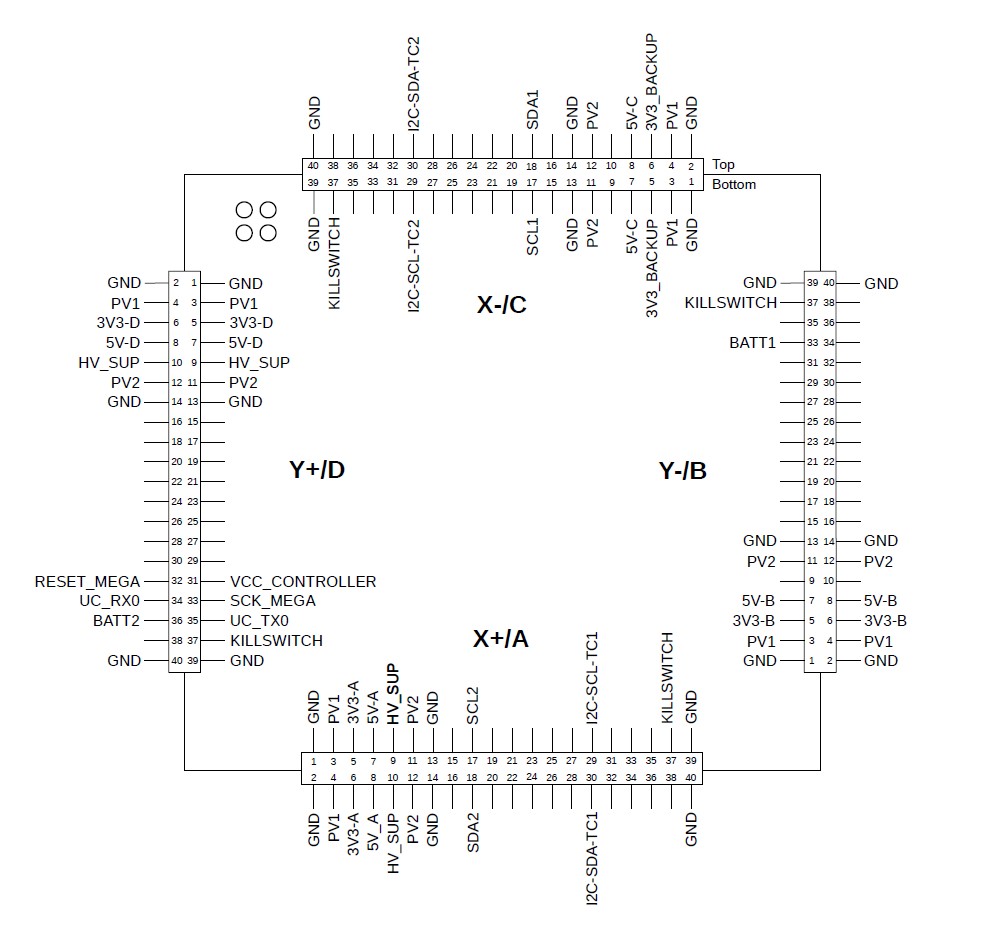


Figure 3-Top View of PSU

A current limit of 200mA is recommended for safety reasons and the voltage setting should be between 6.5V and 9V.

* 1. I²C of CC1/2

CC1: - side X-/C pin 17: SCL1

- side X-/C pin 18: SDA1

CC2: - side X+/A pin 17: SCL2

- side X+/A pin 18: SDA2

Also see **Figure 3.**

1. Programming
   1. Software Interface Atmel Studio/Microchip Studio

To program the MC – Atmega128 the Software Tool “Atmel Studio” or “Microchip Studio” has to be used.

Download Link Atmel Studio: <https://atmel-studio.software.informer.com/7.0/>

Download Link Microchip Studio: <https://www.microchip.com/en-us/tools-resources/develop/microchip-studio>

To program the Atmega128A follow these steps:

* Open the Project file “[EPS\_MC.cproj](https://github.com/DominicRichter/CLIMB_PSU/blob/main/EPS_MC/EPS_MC.cproj" \o "EPS_MC.cproj)”
* In the file “modes.c” you can find the definition of the chosen Debug mode in line and change it.
* Before programming the device open the device manager → **Figure 4**
  + For “Tool” chose the Atmel Ice debugger, and for “Device” enter the correct microcontroller Atmega128A
  + Then klick on “Apply”
  + Afterwards it should be possible to klick on “Read” of the Device Signature
  + If the target voltage and the ID of the microcontroller appears it should be ready to program and the device manager can be closed again

Open device manager

Choose tool and device

Read signature and target voltage

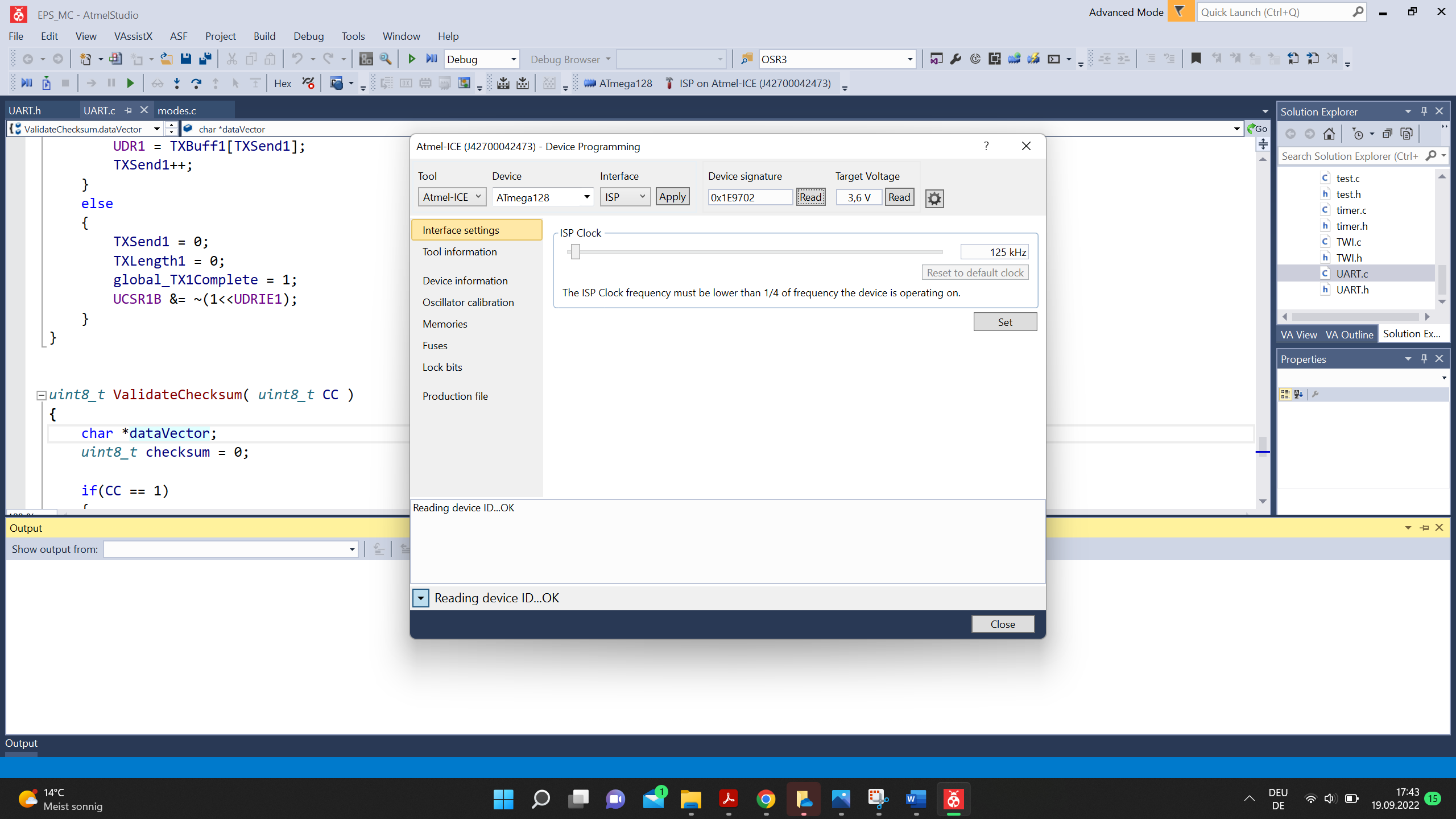


Figure 4-Overview of Atmel Studio Interface

* Afterwards the new Debug Mode can be uploaded by klicking on the green arrow → **Figure 5**

Upload new firmware

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Figure 5-Upload of new firmware

* 1. Overview of Connecting the cables

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Figure 6-Overview connection cables

1. Firmware

The PSU consists of three programmable microchips -> three firmware's. The Main controller (MC) and two communication controllers (CC's). The CC's are the interface between the MC and the OBC. The MC is connected to both CC's and communicates via UART. The CC's communicate with the OBC via I2C (OBC -> Master, CC's -> Slaves). The MC collects housekeeping data by communicating with all sensors via I2C.

* 1. Firmware Main Controller (MC):

The firmware of the main controller is responsible for:

* Converting the power on the PV1 and PV2 bus to 3V3 and 5V.
* Connecting battery to PV1 or PV2 bus.
* Gathering of housekeeping data via I2C (power and temperature sensors).

The firmware holds several modes: power up mode, debug mode, boot mode, flight mode, safe mode, power down mode.

* Power up mode: In this mode the PPU checks if the RBF is LOW or HIGH. If it is LOW, the EPS enters the debug mode and if HIGH enters the boot mode.
* Debug mode: This mode is only for debugging. When entering this mode, the code written in the debug section of the firmware is executed.
* Boot mode: The boot mode switches between flight, safe and power down mode.
* Flight mode: This mode is set by default and provides all functions of the EPS system. In this mode the EPS collects all sensor data, responds to commands, communicates with the communication controllers (CC), manages the batteries, etc.
* Safe mode: The safe mode is entered if the communication to the CC's is lost. In this mode the MC will connect the batteries to the PV1 and PV2 bus and will activate the 3.3V back up power and try to establish communication with the CC's.
* Power down mode: This mode is entered if the battery voltage or temperature is below a certain value. In this mode most power consumers are deactivated, less sensors are used and the communication cycle is reduced. This mode is left when at least one of the batteries is sufficiently charged again.
  1. Firmware Communication Controller 1/2 (CC1/2):

The CC's are the interface between the MC and the OBC. The controllers are in a sleep mode as long no requests occur. The controllers are sending/receiving data between the MC and OBC. The MC creates a data vector with all housekeeping data which is frequently send to the CC1/2 (structure of data vector at end of README). The CC1/2 adds its own housekeeping data (Temperature and Vcc) to the data vector and sends it to the OBC via I2C. The OBC itself sends data to the CC1/2 (e.g. turn on 3V3 on side x+) which are then send to the MC. All I2C communication between the CC and the OBC are handled with ISR's. The controllers are also capable of connecting battery 1 to PV1 and can enable the 3V3\_Backup voltage regulator. The CC2 controller is used as redundancy to the CC1.

Difference between CC1 and CC2:

* The main interface for the OBC is via CC1. Commands which are received via CC1 are handled with higher priority than commands via CC2 and, when conflicting, will be executed.
* CC1 also holds a safe mode which is active if the MC is not available. Not sure if also implemented on CC2?
* CC1 also holds a communication with TT&C but this was only implemented for PEGASUS and is not used for Climb.
  1. How to receive data vector from CC1/2 that is send to OBC:

For the connection to the CC's, there are pins on the PSU for SDA and SCL for both CC's (see chapter 1.3). The address of CC1 is 0x55 and the address of CC2 is 0xAA. To receive the data vector from CC1/2 one has to send the address of one of the CC and then the register number of the data vector. The structure of the data vector is shown at the end of the README. The relevant section in the firmware for I2C communication for CC1 is in the file "InterruptVector\_init.c" beginning from line 836 and for CC2 'twi.c" from line 27 on.

A picture containing graphical user interface

Description automatically generatedExample of an Arduino sketch used to read out the data vector (figure 7):  
CC2\_address = 0xAA; datavector\_reg\_numb = 5;

Figure 7: Arduino example sketch for OBC dummy

The variable "datavector" contains now the 5th byte of the received data vector from CC2.

* 1. Structure of the data vector that is send to the OBC:

In flight mode the MC creates a data vector with all housekeeping data which is frequently send to the CC1/2 and then further to the OBC. The data vector is created in the firmware of the MC with the function "void CreateDataVector(uint8\_t \*dataVector)" which is implemented in the file "CCinterface.c". The vector is structured as follows:

* uint8\_t i;//for checksum
* dataVector[0] = '$';
* dataVector[1] = 'D';
* // 5V VI data
* dataVector[2] = 'P'; //global\_5V.i1Low;
* dataVector[3] = global\_5V.i1High;
* dataVector[4] = global\_5V.i2Low;
* dataVector[5] = global\_5V.i2High;
* dataVector[6] = global\_5V.v1Low;
* dataVector[7] = global\_5V.v1High;
* dataVector[8] = global\_5V.v2Low;
* dataVector[9] = global\_5V.v2High;
* // 3V3 VI data
* dataVector[10] = global\_3V3.i1Low;
* dataVector[11] = global\_3V3.i1High;
* dataVector[12] = global\_3V3.i2Low;
* dataVector[13] = global\_3V3.i2High;
* dataVector[14] = global\_3V3.v1Low;
* dataVector[15] = global\_3V3.v1High;
* dataVector[16] = global\_3V3.v2Low;
* dataVector[17] = global\_3V3.v2High;
* // edge temperature
* dataVector[18] = global\_tEdge.tempLow;
* dataVector[19] = global\_tEdge.tempHigh;
* // center temperature
* dataVector[20] = global\_tCenter.tempLow;
* dataVector[21] = global\_tCenter.tempHigh;
* // HV VI data
* dataVector[22] = global\_HV.i1Low;
* dataVector[23] = global\_HV.i1High;
* dataVector[24] = global\_HV.i2Low;
* dataVector[25] = global\_HV.i2High;
* dataVector[26] = global\_HV.v1Low;
* dataVector[27] = global\_HV.v1High;
* dataVector[28] = global\_HV.v2Low;
* dataVector[29] = global\_HV.v2High;
* // battery 1 VI data
* dataVector[30] = global\_viBat1.i1Low;
* dataVector[31] = global\_viBat1.i1High;
* dataVector[32] = global\_viBat1.i2Low;
* dataVector[33] = global\_viBat1.i2High;
* dataVector[34] = global\_viBat1.v1Low;
* dataVector[35] = global\_viBat1.v1High;
* dataVector[36] = global\_viBat1.v2Low;
* dataVector[37] = global\_viBat1.v2High;
* // battery 2 VI data
* dataVector[38] = global\_viBat2.i1Low;
* dataVector[39] = global\_viBat2.i1High;
* dataVector[40] = global\_viBat2.i2Low;
* dataVector[41] = global\_viBat2.i2High;
* dataVector[42] = global\_viBat2.v1Low;
* dataVector[43] = global\_viBat2.v1High;
* dataVector[44] = global\_viBat2.v2Low;
* dataVector[45] = global\_viBat2.v2High;
* // battery 1 temperature
* dataVector[46] = global\_tBat1.tempLow;
* dataVector[47] = global\_tBat1.tempHigh;
* // battery 2 temperature
* dataVector[48] = global\_tBat2.tempLow;
* dataVector[49] = global\_tBat2.tempHigh;
* // status registers (not correctly implemented yet)
* dataVector[50] = GetEEStatus1();
* dataVector[51] = GetEEStatus2();
* dataVector[52] = GetEEStatus3();
* // estimated capacity battery 1 and 2 (not implemented yet)
* dataVector[53] = 0x01;
* dataVector[54] = 0x02;
* // number of reboots MC, CC1 and CC2 (not implemented yet)
* dataVector[55] = GetEERebootCount();
* dataVector[56] = 0x01;
* dataVector[57] = 0x02;
* // VCC\_CC1, TEMP\_CC1, VCC\_CC2, TEMP\_CC2 (not implemented yet)
* dataVector[58] = 0x01;
* dataVector[59] = 0x01;
* dataVector[60] = 0x02;
* dataVector[61] = 0x02;
* // status CC1, CC2 (not implemented yet)
* dataVector[62] = 0x01;
* dataVector[63] = 0x02;
* // TBD, TBD, CCx FOVR, CCx OVR (not implemented yet)
* dataVector[64] = 0x33;
* dataVector[65] = 0x33;
* dataVector[66] = 0x33;
* dataVector[67] = 0x33;
* // checksum of byte 0 to 65 via XOR
* dataVector[68] = dataVector[0];
* for(i = 1; i <= CHECKSUM\_LIMIT\_MC\_CC; i++)//Checksum
* {
* dataVector[68] ^= dataVector[i];
* }

This data vector with the size of 69 byte is send to the CC1/2. In the firmware of the CC1/2 the vector is extended to the size of 88 byte and filled with data from CC1/2. e.g. In the firmware of CC2 the data vector is extended as follows:

The dataVector from the MC is received from CC2 via UART byte per byte and saved in a new vector called "data".

main.h:

* #define rb\_ct data[55] //reboot counter of
* #define VCC data[58] //supply voltage Vcc
* #define TEMP data[59] //internal temperature
* #define status data[61] //status of CC2
* #define FOVR data[81] //force output value register
* #define OVR data[82] //output value register
* #define new\_data\_flag data[87] //if new data from OBC received since last com. with MC 1, else 0.

uart.c:

* data[56]=uart\_string\_R[58]; //VCC CC1
* data[57]=uart\_string\_R[59]; //temperature of CC1
* data[60]=uart\_string\_R[62]; //status CC1
* //data[62]=uart\_string\_R[64]; //TBD
* data[62]=0xAB; //ID register
* data[63]=uart\_string\_R[65]; //TBD
* data[81]=uart\_string\_R[66]; //CC2 FOVR
* data[82]=uart\_string\_R[67]; //CC2 OVR

The whole new data vector is then send via I2C to the OBC if the I2C interrupt routine is triggered.