DI/OT Monitoring Module (MoniMod)

Christos Gentsos

Sep 12, 2022

Table of Contents

Welcome to the documentation of the DI/OT Monitoring Module (MoniMod). The MoniMod is a PMBus-compatible monitoring module based on a Cortex-M0+ microcontroller, developed as part of the DI/OT project. It can monitor the voltage and current consumption of up to three power rails, system temperature(s), and control up to three fans.

CHAPTER 1

Overview

The DI/OT Monitoring Module (MoniMod) is a monitoring module developed for the DI/OT project's[1] power supply and (optional) fan tray, and based on the ATSAMD21G18 Cortex-M0+ uC[2]. It can monitor voltage and current consumption for up to three power rails, host up to three temperature sensors, and control up to three fans without requiring them to support PWM. The module is accessed and managed through a PMBus interface. A picture of the first prototype can be seen at Fig. 2.1. It will be qualified against radiation effects, up to a total dose of 500 Gy.

1.1 Repository Structure

The project's repository[3] is intended to be as complete as possible, containing both the PCB design and the uC firmware source.

The MoniMod PCB design can be found in the `monimod-pcb` directory. The firmware sources are currently split in three separate programs:

1. The main FW, that implements most functionality. This lives in the `main_fw` directory.
2. The bootloader, that implements remote programming. That lives in the `bootloader` directory.
3. The simple I2C master written to help develop the main FW can be found in the `test_master` directory.

Beside these three program directories, there is a `common` directory that hosts shared code, and a `utils` directory that is used to host any general development utilities. There is also a `build` directory which hosts a very simple top-level makefile; this simply builds all FW binaries and gathers them in one place.

Each program project has an `atmel_start_prj` subdirectory: this is a testament to the use of the Atmel START tool[4] to generate drivers, linker scripts and makefiles. The structure of the generated files has been slightly altered, with our code located in the `src`, `include` and `build` subdirectories, outside of `atmel_start_prj`.

1.2 Feature Checklist

The following features have been implemented:

[1] DI/OT project home: https://www.ohwr.org/project/diot/wikis/home
[2] ATSAMD21 product page: https://www.microchip.com/wwwproducts/en/ATsamd21g18
[3] MoniMod repository: https://www.ohwr.org/project/diot-monimod
[4] ATMEL START: https://start.atmel.com

- Option to use USB as a terminal to print debug messages and possibly interact with the program
- Implement the PMBus command subset for voltage, current and temperature monitoring (by means of a LM61 sensor)
- Implement the PMBus command subset for fan control and monitoring
- Support Packet Error Checking (PEC) for robustness (as described in the SMBus specification[6])
- PID fan control
- Versatile temperature control, configured using manufacturer specific PMBus commands
- Per-command callback support on command writes and reads
- Support for reset over manufacturer specific PMBus command
- Bootloader support for remote reprogramming over manufacturer specific PMBus commands
- Radiation mitigation measures

[6] The SMBus 2.0 specification: http://smbus.org/specs/smbus20.pdf

CHAPTER 2

PCB

To draw the schematics and the layout of the 4-layer PCB, the open-source KiCad suite[1] was used. The first prototype PCB has been built (see Fig. 2.1) and tested; as the thin purple wires attest, a number of bugs have been spotted and fixed as a result.

Note: The dimensions of this prototype are 1.53in $\times$ 3.14in. Although this is already quite compact, half of that space is dedicated to the fan driving circuitry. Consequently, a separate revision without fan driving capabilities is planned to ease the integration with the passively-cooled PSU[3]. Also, in case PT100-type temperature sensors are not eventually used, further size reductions are expected.

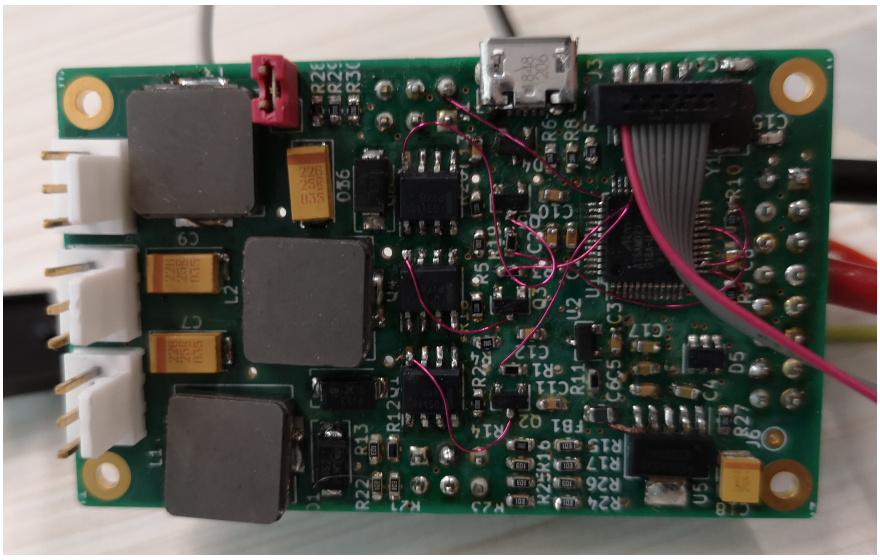

Fig. 2.1: The first MoniMod prototype

[1] KiCad EDA home page: http://www.kicad-pcb.org/
[3] RaToPUS home: https://ohwr.org/project/psu-rad-acdc-230v-12v5v-110w/wikis/home

2.1 Main blocks

2.1.1 Linear Regulator

The MoniMod is powered by a 5V rail, but the uC needs a power supply between 1.62V and 3.63V to operate; on the other side, SMBus (which defines the electrical characteristics of PMBus) has a power supply range of 3V to 5V ($\pm$ 10%). To satisfy these constraints, the uC can be operated at 3.3V.

Since its current consumption can be quite low (found to be ~40–45mA) it is enough to use a simple linear regulator to generate this power rail: the TPS7A4533 has been selected from[2].

2.1.2 SAMD21 uC

To power the uC, 1uF and 0.1uF bypass capacitors are placed close to the digital power supply pins; a ferrite bead is used to decouple the 3.3V analog domain from the noise in the digital one.

The uC clock is provided by a 8MHz crystal.

A Micro USB Type B connector allows one to use the USB peripheral of the uC for firmware debugging reasons; a TVS array protects the device from any ESD events. Also, a SWD (Serial Wire Debug) interface is exposed in an on-board header and an external connector.

2.1.3 Fan drivers

Fan driving circuitry has been designed to modulate the DC voltage of the fans such that speed control can be exercised on non-PWM capable models. To convert the 25kHz PWM signals from the uC to DC levels, a buck topology has been used (see Fig. 2.2).

Components selection allows using 12V fans at 1A maximum current; this is the reason for the quite bulky inductors that can be seen in Fig. 2.1. Also, this topology might allow a large inrush current if the PWM duty cycle were to change too rapidly; this is handled in the software, which limits the rate of change of the duty cycle.

Note: In the v1 prototype a PMOS device was used, with a BJT driving it. This has been replaced due to the lack of suitable (low Ron) devices that are also radiation-tested.

Note: The low-side driver is not needed here but it is used in the RaToPUS power supply: we use common components where possible because of the high cost associated with qualifying them.

2.1.4 Analog inputs

Due to an ADC and an analog MUX being integrated in the uC and the benefit of having as few components as possible (less points of failure in radiation), the analog frontend is kept simple; its basic building blocks are documented below.

Voltage sensing

For the voltage inputs, an impedance of the order of some 10ks is high enough to keep the quiescent current negligible, and since this is designed to measure levels in the system PSU and not some ultralow noise power supply, it is also sufficient to decouple the potentially noisy switching capacitor analog MUX inside the uC from the signal source.

[2] CERN radiation test database: https://radwg-table.web.cern.ch/public/

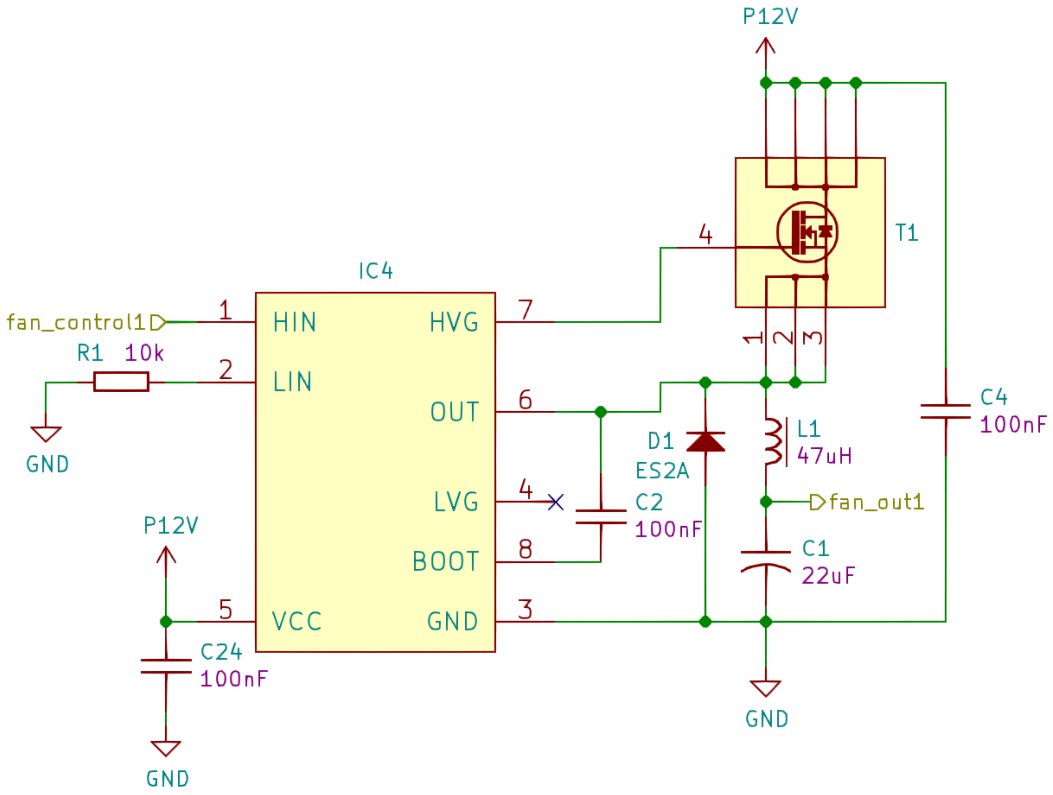

Fig. 2.2: Schematic of the fan driver buck circuit

Current sensing

The current sensing uses the differential amplifier in Fig. 2.3 to lower the voltage difference in the sense resistor (on the host board) and amplify it to a level suitable for sampling with the ADC.

The primary degradation in BJTs exposed in radiation is a reduction in their forward current gain; in this circuit the opamp can easily provide the extra current so the performance should not degrade too much over time.

Temperature sensing

In the v2 prototype version, the user can switch between LM61-type and PT100 / PT1000 temperature sensors using the three switches in the back side of the board. This allows comparing the two different solutions under the same conditions (under radiation) and making an informed decision on which type to use in the next version.

As seen on Fig. 2.4, when the user selects the PT100 option the switch connects a current reference implemented with the Vref and one half of an AD8030 rail-to-rail opamp. The same node is connected to a x15 amplifier, implemented using the other half of the opamp. When the user selects LM61 operation, the switch disconnects the current source and the amplifier such that the sensor input is connected directly to the ADC input.

2.2 Connections

The interface between the MoniMod and the system board is contained in *connector J2*. A limit of 1A per pin requires the use of multiple pins for the 12V supply, since at full speed the fans could draw up to 3A of current.

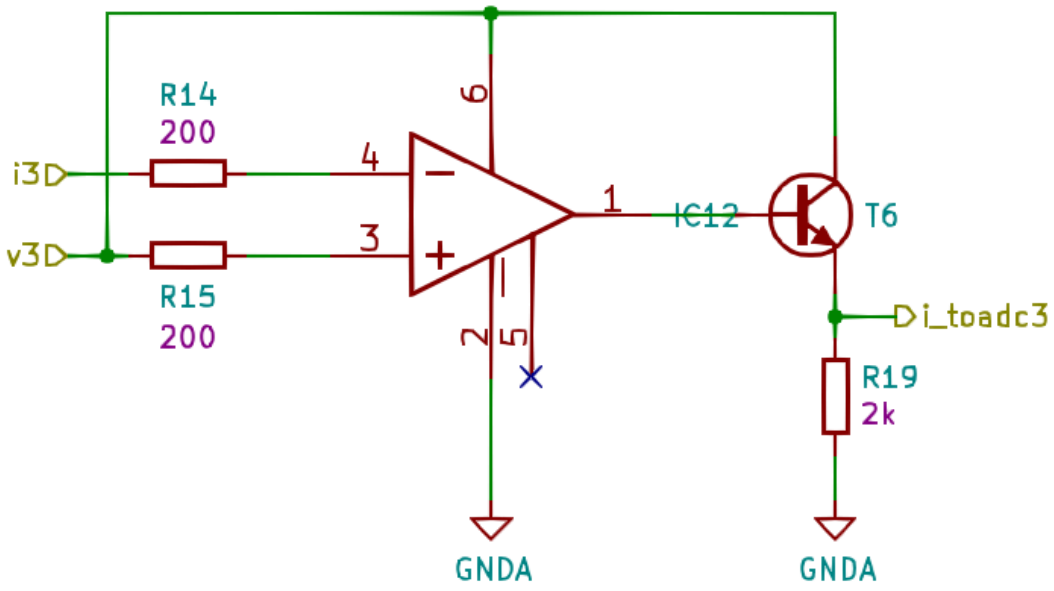

Fig. 2.3: Schematic of the current sensing circuit

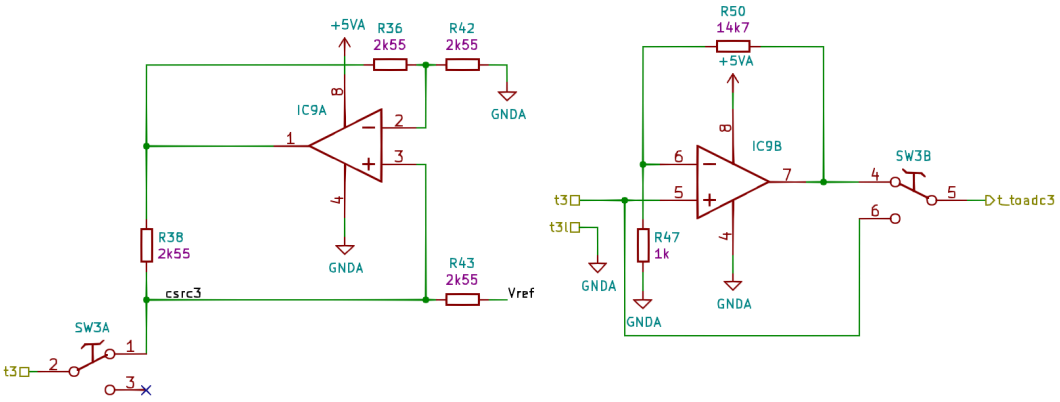

Fig. 2.4: Schematic of the temperature sensing circuit

Table 2.1: J2 pinout

pin	name	pin	name
1	GND	2	GND
3	RST_N	4	P12V
5	M_SCL	6	GND
7	P5V	8	P12V
9	M_SDA	10	GND
11	M_IO1	12	P12V
13	M_IO2	14	GND
15	GND	16	P12V

Connector J3 can be used to program and debug the uC using an SWD debugger (like the J-Link EDU Mini).

Table 2.2: J3 pinout

pin	name	pin	name
1	P3V3	2	PGM_SWDIO
3	GND	4	PGM_SWCLK
5	GND	6	N/C
7	N/C	8	N/C
9	N/C	10	PGM_RST_N

Connectors J4 and *J5* are used to configure the PMBus address the MoniMod will assume, and to connect it to what it is monitoring: the temperature sensors; the voltage rails; and the current sense outputs.

Note: In the current version, the current sense just reads an absolute voltage; the next revision shall include a simple opamp-based circuit to read a standard high-side sense resistor.

Table 2.3: J4 pinout

pin	name	pin	name
1	TMP1	2	TMP1_N
3	TMP2	4	TMP2_N
5	TMP3	6	TMP3_N

Table 2.4: J5 pinout

pin	name	pin	name
1	V1	2	I1
3	V2	4	I2
5	V3	6	I3

Table 2.5: J6 pinout

pin	name
1	ADDR0
3	ADDR1
5	ADDR2

2.3 Radiation qualification

Since the board will need to be qualified against radiation effects, it helps if it is comprised of components that are already qualified; for that reason, parts from the CERN radiation test database were preferred. Table 2.6 shows the list of active

components, along with the tests that have been performed for each one. Some possible deployment locations, along with their radiation levels, are listed on Table 2.8.

Table 2.6: Radiation data for active components

Name	Function	EDMS Rad Test #	Rad Test Type
TL431BQDBZ	Reference Voltage	1171338	TID, SEE
TPS7A4533DCQ	Linear Regulator	1911630	TID, SEE
AD8030ARZ	Dual rail-to-rail opamp	1729078	TID, SEE
AD8029AKSZ	rail-to-rail opamp	1729078	TID, SEE
IRFH5025TRPBF	N-channel Power MOSFET	2207602	TID, SEE
BC817-25	npn BJT	1583307	TID
ATSAMD21G18A	uC	N/A	N/A
L6498D	Gate driver	N/A	N/A

Note: The uC report hasn't been uploaded to the radiation database yet but it has been tested, with encouraging results.

Note: The gate driver has not been tested yet, but a number of similar components is scheduled to be tested. In case the selected component proves to be unsuitable, we can be confident that there will be another, similar one, that will replace it at the next version of the PCB.

On the other hand, some components, such as TVS diodes, do not degrade in a way that would impact the design. As such, they have been selected freely. These are listed on Table 2.7.

Table 2.7: Active components that don't need radiation testing

Name	Function
ES2A	Fast Recovery Diode
PESD24VS5UD	TVS Diodes
PESD5V0S2UAT	TVS Diodes

Table 2.8: HL-LHC radiation levels for various locations[4]

Location	Dose [Gy/y]	HEH [pp / cm ^{2} / y]	1MeV [n / cm ^{2} / y]
UJ	10	5e9	5e10
UL	0.2	1e8	1e9
RR alcoves	6	3e9	3e10
ARC	2	1e9	1e9

[4] LHC and HL-LHC: Present and Future Radiation Environment in the High-Luminosity Collision Points and RHA Implications: https://ieeexplore.ieee.org/document/8116686

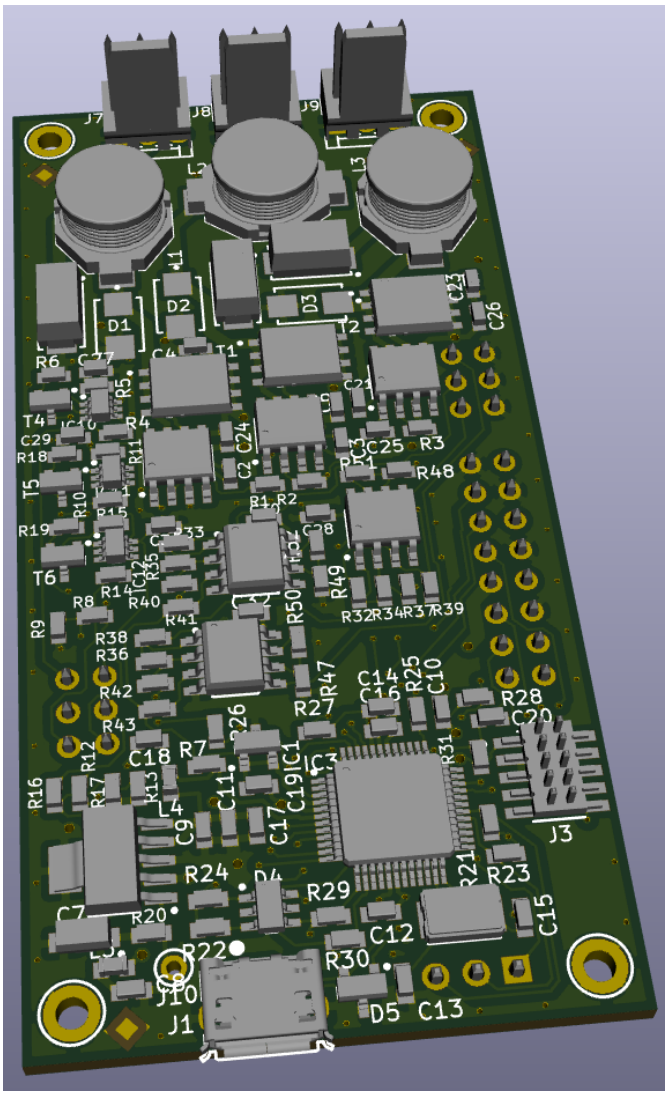

Fig. 2.5: A rendering of the second MoniMod prototype

CHAPTER 3

Firmware

The project's fitmware is split in three parts: the bootloader, the main firmware and the test firmware.

3.1 PMBus command infrastructure

A common command handling infrastructure has been put in place, such that both the main firmware and the bootloader can easily implement different subsets of PMBus commands. The basic construct of this implementation is the `cmd_t` structure:

struct cmd_t

Public Members

const uint8_t **addr**
CMD code.

int8_t ***const data_len**
transaction length for this command

uint8_t ***const data_pnt**
pointer to data

const fp_t **a_callback**
invoked when accessing the command, before any data transfer

const fp_t **w_callback**
invoked after writing data

const fp_t **r_callback**
invoked after reading data

const uint8_t **query_byte**
data for the query command

const uint8_t **wr_pec_disabled**
always disable PEC for this command if non-zero

An array of these structs makes up a command space:

struct cmd_space_t

Public Members

const uint8_t **n_cmds**
holds number of commands implemented

cmd_t ***const cmds**
where the command structure list is stored

From the user's point of view, these structures are defined and used just once, in the function

void **setup_I2C_slave**(*cmd_space_t *impl_cmds*)

This function will configure the inturrupt handlers, below, with the command spaces defined in the specific user implementation (main firmware or bootloader). From that point on, the only interaction will be through the user-defined callbacks.

```
static void __xMR I2C_rx_complete(const struct i2c_s_async_descriptor *const descr)
static void __xMR I2C_tx_pending(const struct i2c_s_async_descriptor *const descr)
static void __xMR I2C_tx_complete(const struct i2c_s_async_descriptor *const descr)
static void __xMR I2C_error(const struct i2c_s_async_descriptor *const descr)
```

3.2 Bootloader

The bootloader, after bringing up the device, will check for the special word `0xBEC0ABCD` in the flash storage (see struct below) and, depending on the value, will either hand control to the main FW, or enter remote programming (bootloader) mode.

struct user_flash_t
This struct defines 256 bytes of user data, stored in non-volatile memory, including a special 4-byte word which is used to turn on remote programming.

Public Members

uint32_t **copy_fw**
check if we want to enable the remote programming functionality

uint16_t **setfrpms**[3]
store fan configuration and speeds

uint8_t **user_data**[228]
provide some (optional) user data storage

3.3 PMBus commands overview

The full list of PMBus commands implemented by the MoniMod can be found in Table 3.1 and Table 3.2.

All physical quantities (except output voltage, which is discussed below) are expressed in the 16-bit PMBus Linear data format (LINEAR11, Fig. 3.1), instead of the (somewhat more complex) Direct format PMBus also supports. An 11-bit mantissa (Y) and a 5-bit exponent (N), expressed in 2's complement, form a floating-point number X according to $X = Y \cdot 2^N$.

The output voltages use a different, 21-bit format, called LINEAR16 (in contrast to the 16-bit LINEAR11, the names are derived from the mantissa width). This format comprises a 5-bit 2's complement exponent, reported by the 5 MSBs of the *VOUT_MODE* command; and a 16-bit unsigned mantissa, reported by *READ_VOUT*. Many COTS PSUs use LINEAR11 for everything, and this behavior is also possible using a compile-time switch. In the MoniMod, the exponential factor for the LINEAR16 format is fixed to -10, so the voltages reported can be obtained with $X = Y \cdot 2^{-}10$, where Y is the 16-bit value returned by *READ_VOUT*.

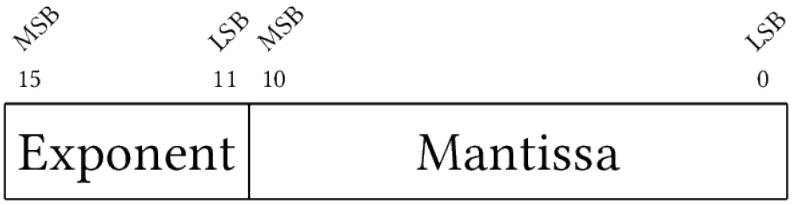

Fig. 3.1: The PMBus Linear data format

Table 3.1: PMBus commands implemented by the MoniMod

Cmd code	Command name	Transaction type	Data len	Description
00 +	*PAGE*	Byte write / read	1	set get page
19 +	*CAPABILITY*	Byte read	1	capabilities of the device
1A +*	*QUERY*	Block w / r proc. call	1	query cmd props
20 +	*VOUT_MODE*	Byte read	1	read voltage format
3A +	*FAN_CONFIG_1_2*	Byte write / read	1	config fans 1&2
3B +	*FAN_COMMAND_1*	Word write / read	2	set fan 1 speed
3C +	*FAN_COMMAND_2*	Word write / read	2	set fan 2 speed
3D +	*FAN_CONFIG_3_4*	Byte write / read	1	config fan 3
3E +	*FAN_COMMAND_3*	Word write / read	2	set fan 3 speed
78 +	*STATUS_BYTE*	Byte read	1	status byte
7E +	*STATUS_CML*	Byte read	1	status CML
8B +	*READ_VOUT*	Word read	2	read voltage
8C +	*READ_IOUT*	Word read	2	read current
8D +	*READ_TEMPERATURE_1*	Word read	2	read temp. sensor 1
8E +	*READ_TEMPERATURE_2*	Word read	2	read temp. sensor 2
8F +	*READ_TEMPERATURE_3*	Word read	2	read temp. sensor 3
90 +	*READ_FAN_SPEED_1*	Word read	2	read fan 1 speed
91 +	*READ_FAN_SPEED_2*	Word read	2	read fan 2 speed
92 +	*READ_FAN_SPEED_3*	Word read	2	read fan 3 speed
96 +	*READ_POUT*	Word read	2	read power
99 +*	*MFR_ID*	Block read	var	manufacturer ID
9A +*	*MFR_MODEL*	Block read	var	model
9B +*	*MFR_REVISION*	Block read	var	revision
9C +*	*MFR_LOCATION*	Block read	var	location
9D +*	*MFR_DATE*	Block read	var	date
9E +*	*MFR_SERIAL*	Block read	var	serial number
AE +*	*IC_DEVICE_REV*	Block read	var	git commit id

Commands marked with + or * are:

+ - supported by main fw

* - supported by bootloader

Table 3.2: Manufacturer specific PMBus commands implemented by the MoniMod

Cmd code	Command name	Transaction type	Data len	Description
D1 *	*WRITTEN_FW_SIZE*	Word write / read	2	size of the FW to be written
D2 *	*WRITTEN_FW_BLOCK*	MultiByte write / read	8	FW block to be written
D3 *	*WRITTEN_FW_CHKSUM*	Word write / read	2	checksum of the written FW
D4 *	*LOCAL_FW_CHKSUM*	Word write	2	calculated checksum of the written FW
D5 +*	*BOOT_NEW_FW*	Byte write / read	1	on write turn on btldr pgm mode, reset; on read get which fw is running
D6 +*	*UC_RESET*	Byte write	1	reset the uC
D7 +	*UPTIME_SECS*	Block read	var(1+4)	get the uptime in seconds
D8 +	*TMR_ERROR_CNT*	Block write / read	var(1+4)	clear / get TMR error count
D9 +	*USE_PEC*	Byte write / read	1	turn PEC on / off
E0 +	*TEMP_CURVE_POINTS*	Block write / read	var(1+13)	set / get temp. curve points
E1 +	*TEMP_MATRIX_ROW*	Block write / read	var(1+7)	set / get temp. matrix points
E2 +	*TC_ONOFF*	Byte write / read	1	turn temp. control on / off

Commands marked with + or * are:

+ - supported by main fw

* - supported by bootloader

Block registers contain in the first byte the size of data that follows. This also applies to the registers with the fixed size like *UPTIME_SECS*.

3.4 Detailed list of PMBus commands

3.4.1 PAGE

Command code: **00**
Transaction type: **Byte write / read**
Data length: **1**

The PAGE command is used to select a power rail for the *READ_VOUT*, *READ_IOUT* and *READ_POUT* commands. Allowed values for the page parameter are $0 \leq N \leq 3$ for the DI/OT Rad-Tol System Board revision and $0 \leq N \leq 2$ for the fan-tray, RaToPUS and generic prototype revisions.

3.4.2 CAPABILITY

Command code: **19**
Transaction type: **Byte read**
Data length: **1**

The CAPABILITY command returns one byte of information with some key capabilities of a PMBus device.

Table 3.3: CAPABILITY Data byte format

Bit(s)	Meaning
7	Packet Error Checking is supported
6:0	Unused

Bit 7 of the CAPABILITY register reflects the use of PEC set by the *USE_PEC* register.

3.4.3 QUERY

Command code: **1A**
Transaction type: **Block w / r proc. call**
Data length: **1**

The QUERY command takes a command code as an argument and replies with information on the command: whether it is supported, if read or write is supported, and what data format it works with.

3.4.4 VOUT_MODE

Command code: **20**
Transaction type: **Byte read**
Data length: **1**

The VOUT_MODE command reports the format the device uses for measured voltage related data. The 3 MSBs indicate whether that's *Linear* (0b000), VID (0b001) or Direct (0b010), and in the MoniMod's case it's always 0b000 for Linear format. The 5 MSBs return either 0x16, for LINEAR16 format used (fully PMBus-compliant operation, fixed $2^{-}10$ exponential); or 0x00, for LINEAR11 format used (common for COTS PSUs). See *Linear* for more details on the specifics of these formats.

3.4.5 FAN_CONFIG_n_m

Command codes: **3A, 3D**
Transaction type: **Byte write / read**
Data length: **1**

The FAN_CONFIG_1_2 and FAN_CONFIG_3_4 commands are used to configure the fans at positions 1, 2, and 3. The format of the configuration byte can be seen in Table 3.4. The two bits that set the tachometer pulses / revolution, which take the values 0–3, correspond to 1–4 pulses per revolution.

Table 3.4: FAN_CONFIG_1_2 and FAN_CONFIG_3_4 data byte format

Bit(s)	Value	Meaning
7	1	Fan 1 / 3 installed
	0	Fan 1 / 3 not installed
6	1	Fan 1 / 3 commanded in RPM
	0	Fan 1 / 3 commanded in duty cycle
5:4	0–3	Fan 1 / 3 tachometer pulses / rev
3	1	Fan 2 installed
	0	Fan 2 not installed
2	1	Fan 2 commanded in RPM
	0	Fan 2 commanded in duty cycle
1:0	0–3	Fan 2 tachometer pulses / rev

3.4.6 FAN_COMMAND_n

Command code: **3B, 3C, 3E**
Transaction type: **Word write / read**
Data length: **2**

The FAN_COMMAND_n commands set the desired speed of the attached fans. The value set is either in RPMs (when the fan is configured to be controlled like that) or duty cycle, in the range 0–1000.

3.4.7 STATUS_BYTE

Command code: **78**
Transaction type: **Byte read**
Data length: **1**

The STATUS_BYTE command returns one byte of information with a summary of the most critical faults.

Table 3.5: STATUS_BYTE Data byte format

Bit(s)	Meaning
7:2	Unused
1	A communications, memory or logic fault has occurred
0	Unused

The value of STATUS_BYTE is calculated based on other status registers (for now only STATUS_CML). To clear value of STATUS_BYTE, please clear information in other status registers.

3.4.8 STATUS_CML

Command code: **7E**
Transaction type: **Byte write / read**
Data length: **1**

The STATUS_CML command returns one data byte with contents as follows:

Table 3.6: STATUS_CML Data byte format

Bit(s)	Meaning
7:6	Unused
5	Packet Error Check Failed
4	TMR Error
3:2	Unused
1	A communication fault
0	Unused

Write 1's in the desired bits positions to clear corresponding errors in the register.

Clearing TMR Error flag sets *TMR_ERROR_CNT* register to 0.

3.4.9 READ_VOUT

Command code: **8B**
Transaction type: **Word read**
Data length: **2**

The READ_VOUT command is used to get the measured voltage of the rail indicated by the last PAGE command (by default that would be the first one).

3.4.10 READ_IOUT

Command code: **8C**
Transaction type: **Word read**
Data length: **2**

The READ_IOUT command is used to get the measured current of the rail indicated by the last PAGE command (by default that would be the first one).

3.4.11 READ_TEMPERATURE_N

Command code: **8D, 8E, 8F**
Transaction type: **Word read**
Data length: **2**

The READ_TEMPERATURE_n commands return the measured temperature from the three installed temperature sensors.

3.4.12 READ_FAN_SPEED_N

Command code: **90, 91, 92**
Transaction type: **Word read**
Data length: **2**

The READ_FAN_SPEED_n return the fan speed of an installed fan, or 0 in case no fan is installed in the pertinent location.

3.4.13 READ_POUT

Command code: **96**
Transaction type: **Word read**
Data length: **2**

The READ_POUT command is used to get the measured power of the rail indicated by the last PAGE command (by default that would be the first one).

3.4.14 MFR_ID

Command code: **99**
Transaction type: **Block read**
Data length: **var**

This returns the manufacturer ID string, “CERN (BE/CO)”.

3.4.15 MFR_MODEL

Command code: **9A**
Transaction type: **Block read**
Data length: **var**

This returns the manufacturer model string, “DI/OT MoniMod”.

3.4.16 MFR_REVISION

Command code: **9B**
Transaction type: **Block read**
Data length: **var**

This returns the manufacturer revision string.

3.4.17 MFR_LOCATION

Command code: **9C**
Transaction type: **Block read**
Data length: **var**

This returns the manufacturer ID string, “Geneva”.

3.4.18 MFR_DATE

Command code: **9D**
Transaction type: **Block read**
Data length: **var**

This returns the manufacturer date string, which currently corresponds to the date of the last release (and not the build used, for example).

3.4.19 MFR_SERIAL

Command code: **9E**
Transaction type: **Block read**
Data length: **var**

This returns a manufacturer serial string (currently unused, returns “123456789”).

3.4.20 IC_DEVICE_REV

Command code: **AE**
Transaction type: **Block read**
Data length: **var**

This returns a git commit id of the used firmware.

3.4.21 WRITTEN_FW_SIZE

Command code: **D1**
Transaction type: **Word write / read**
Data length: **2**

Before writing a new FW binary through the bootloader, its size in bytes divided by 8 has to be given using this command. The written value can be read.

3.4.22 WRITTEN_FW_BLOCK

Command code: **D2**
Transaction type: **MultiByte write / read**
Data length: **8**

A new binary is written to the bootloader in consecutive chunks of 8 bytes, using this command. The written data can be read.

3.4.23 WRITTEN_FW_CHKSUM

Command code: **D3**
Transaction type: **Word write / read**
Data length: **2**

After setting the size of the FW binary with WRITTEN_FW_SIZE and writing it with the WRITTEN_FW_BLOCK command, its SYS-V checksum should be written with this command. This command also resets the write pointers. The written checksum can be read. Please note that this is not the calculated checksum. The calculated checksum is stored in LOCAL_FW_CHKSUM (Section 3.4.24).

3.4.24 LOCAL_FW_CHKSUM

Command code: **D4**
Transaction type: **Word read**
Data length: **2**

After writing the new firmware, this command can be used to get the checksum calculated by the MoniMod. Please note that it is up to the writer to verify (compare) the checksum.

3.4.25 BOOT_NEW_FW

Command code: **D5**
Transaction type: **Byte write / read**
Data length: **1**

The BOOT_NEW_FW command changes the execution mode of the Firmware. When the byte 0xAD is written to this register the running firmware is switched between bootloader and main mode. When the proper byte is received, a special code is written to or removed form the flash memory to let the bootloader to stay in bootloader mode or proceed to the main firmware. More information is available in the section *Bootloader*.

The read gives the information which firmware is actually running.

Value	Meaning
1	Bootloader
2	Main firmware

3.4.26 UC_RESET

Command code: **D6**
Transaction type: **Byte write**
Data length: **1**

Writing 0x5A byte to this register triggers a uC reset. Other writes are silently ignored.

3.4.27 UPTIME_SECS

Command code: **D7**
Transaction type: **Block read**
Data length: **var(1+4)**

Get the uptime of the MoniMod (in seconds).

The fist byte of the register contains the number of bytes of data. For this register, its value is fixed to 4.

3.4.28 TMR_ERROR_CNT

Command code: **D8**
Transaction type: **Block write / read**
Data length: **var(1+4)**

When software mitigation through COAST is enabled (see *TMR using COAST*), one can access the TMR_ERROR_CNT counter using this command. Setting *TMR Error* bit in *STATUS_CML* clears the value of this register.

It is possible to clear the TMR Error counter by writing 0's as data to this register.

The fist byte of the register contains the number of bytes of data. For this register, its value is fixed to 4.

3.4.29 USE_PEC

Command code: **D9**
Transaction type: **Byte write / read**
Data length: **1**

The SMBus specification indicates that a device's PEC support could be enabled or disabled at will.

The read gives the information whether the PEC is enabled.

Value	Meaning
0x00	PEC disabled
0x01	PEC enabled

To change the state of the PEC, the proper magic value has to be written to this register. All other values are silently ignored.

Value	Meaning
0x0F	Disable PEC
0x37	Enable PEC

The command itself is used without a PEC byte appended, no matter whether the function is enabled or not.

3.4.30 TEMP_CURVE_POINTS

Command code: **E0**
Transaction type: **Block write / read**
Data length: **var(1+13)**

As described in the *Temperature control* section, the temperature curve can be set separately for each fan. To do this, the format in Fig. 3.2 has to be used.

0	1	2	3	4	5	6	7	8	9	10	11	12
FAN_N	$T_{0,L}$	$T_{0,H}$	$S_{0,L}$	$S_{0,H}$	$T_{1,L}$	$T_{1,H}$	$S_{1,L}$	$S_{1,H}$	$T_{2,L}$	$T_{2,H}$	$S_{2,L}$	$S_{2,H}$

Fig. 3.2: Temperature curve data frame

The first byte specifies the fan number (0-2), following bytes contains data specific for temperature curve.

The first read after the write to this register allows to get just stored data for the particular fan. The following reads of this register iterates over all fans and allow to get their data.

The fist byte of the register contains the number of bytes of data. For this register, its value is fixed to 13.

3.4.31 TEMP_MATRIX_ROW

Command code: **E1**
Transaction type: **Block write / read**
Data length: **var(1+7)**

As described in the *Temperature control* section, the temperature matrix can be set separately for each fan. The data format for the operation is illustrated in Fig. 3.3.

0	1	2	3	4	5	6
FAN_N	$m_{0N,L}$	$m_{0N,H}$	$m_{1N,L}$	$m_{1N,H}$	$m_{2N,L}$	$m_{2N,H}$

Fig. 3.3: Temperature matrix data frame

The first byte specifies the fan number (0-2), following bytes contains data specific for temperature control.

The first read after the write to this register allows to get just stored data for the particular fan. The following reads of this register iterates over all fans and allow to get their data.

The fist byte of the register contains the number of bytes of data. For this register, its value is fixed to 7.

3.4.32 TC_ONOFF

Command code: **E2**
Transaction type: **Byte write / read**
Data length: **1**

Using the TC_ONOFF command with a zero argument disables Temperature Control, while any non-zero value enables it.

3.5 Fan control PID

When the fans connected provide a tachometer output, fan speed control can be enabled. This is implemented using PID controllers, with each fan having its own instance. The main data structure of the PID implementation is

`struct pid_cntrl_t`

Public Members

float **`setpoint`**
controller setpoint

float **`last_input`**
the input of the last timestep

float **`output_sum`**
storage for integration

uint16_t **`id_cnt`**
timestep counter

This is used by the main software to set the PID setpoint, and by the PID controller to hold integration data. The main function that has to be called every timestep is described below:

float **pid_compute** (*pid_cntrl_t* *pid_inst, float *input*)

use this function with a PID structure and an input to calculate the output for each timestep.

Compute the PID output for the next timestep

Return the PID controller output

Parameters

- `pid_inst`: struct that holds the PID controller's configuration
- `input`: the current input to the PID controller

3.6 Temperature control

The MoniMod implements a very flexible temperature control scheme. Each fan can be assigned its own 3-point temperature–speed curve, as in Fig. 3.4. Temperatures outside the set range will adopt the speed of the minimum and maximum temperature, accordingly.

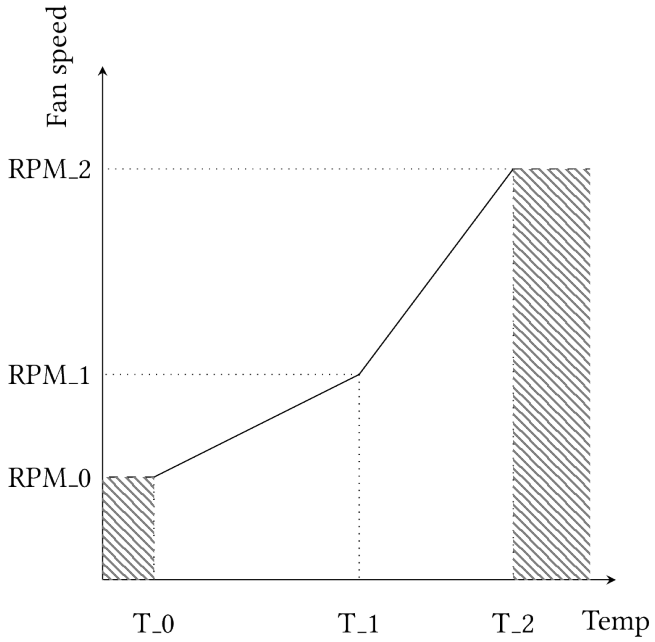

Fig. 3.4: Temperature curve

Moreover, the temperature each fan considers for its curve is a weighted product of all three monitored temperatures, as in Fig. 3.5. This allows one to easily configure the MoniMod to match a wide variety of fan / sensor setups, e.g.:

- each fan is assigned its own temperature sensor
- all three temperatures are averaged to give a more precise system temperature
- one fan blows directly on a sensitive component which is monitored, the other two fans handle the rest of the system

3.7 Test firmware

To help with development, a test firmware has been written for a Feather M0 Basic minimal development board.

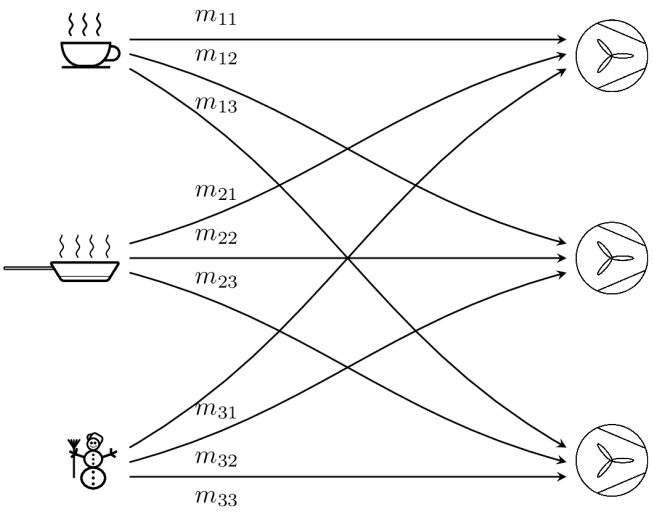

Fig. 3.5: Temperature mixing matrix

3.8 Mitigation measures

The MoniMod will be used in radiation environments. Although its function is not critical and it can be remotely reset upon loss of communication, some measures have been taken to minimize interruptions and data corruption, leading to an improved QoS.

3.8.1 TMR using COAST

The COAST LLVM passes can be optionally used to automatically implement TMR (Triple Modular Redundancy) in important and long-lived variables. This can particularly benefit the integrity of dynamic configuration data that gets set in the memory once and then gets read periodically, or state machines such as the one in the I2C interrupt handlers which is critical for stable communications.

3.8.2 NOPs and trampolines

The Program Counter is also sensitive to SEUs; in fact, execution can sometimes jump to an invalid address. To help mitigate failures owed to this mechanism, any region of unused memory space has been filled with `NOP` instructions, and a small trampoline function as an epilogue that will reset the stack pointer and jump to the device initialization code. Furthermore, the instruction that comprises the main loop has been placed at a "strategic" location, aligned by `0x8000`: that way, a bit-flip in any of the lower bits will send execution to the upper memory region, filled with the NOPs and concluding at the trampoline.

3.8.3 Watchdog

The uC integrates a watchdog peripheral: this is fed every time the main timer callback runs, i.e. every 10ms. The watchdog is set to trigger if it doesn't get fed for 20ms – as soon as the main loop skips a beat. That ensures a quick revival of the uC and should lead to minimal downtime.

3.8.4 Bling scrubbing

Blindly scrubbing the configuration of peripherals can be used to reduce gradual corruption of their configuration during operation. The frequency has to be carefully selected to minimize downtime.

Note: This hasn't been implemented yet, this is a reminder to do it.

3.8.5 Stack protection

The compilers' stack protection feature is enabled to catch the corner case that some loop goes awry and corrupts the stack due to some SEU. In case that happens, the uC quickly gets reset.

3.9 Toolchains

The project can be built with GCC and Clang / LLVM compilers; one can switch between the two simply by setting a Makefile variable. Note, however, that TMR only works with Clang.

Index