DI/OT Monitoring Module (MoniMod)

Christos Gentsos

Jun 29, 2020

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.

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 extended commands
- Per-command callback support on command writes and reads
- Extended commands compatible with the PMBus specification[5]
- Support for reset over extended PMBus commands
- Bootloader support for remote reprogramming over extended PMBus commands
- Radiation mitigation measures

[6] The SMBus 2.0 specification: http://smbus.org/specs/smbus20.pdf
[5] The PMBus 1.0 specification, part II: http://pmbus.org/Assets/PDFS/Public/PMBus_Specification_Part_II_Rev_1_0_20050328.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: It is worth noting here that the components used in this first prototype are not the ones that will be used in the next one; only a few have been picked out of the CERN radiation test database[2], whereas all components will be radiation-tested in the next version.

Note: The dimensions of this prototype are 63mm × 38mm. 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].

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 (± 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.

[1] KiCad EDA home page: http://www.kicad-pcb.org/
[2] CERN radiation test database: https://radwg-table.web.cern.ch/public/
[3] RaToPUS home: https://ohwr.org/project/psu-rad-acdc-230v-12v5v-110w/wikis/home

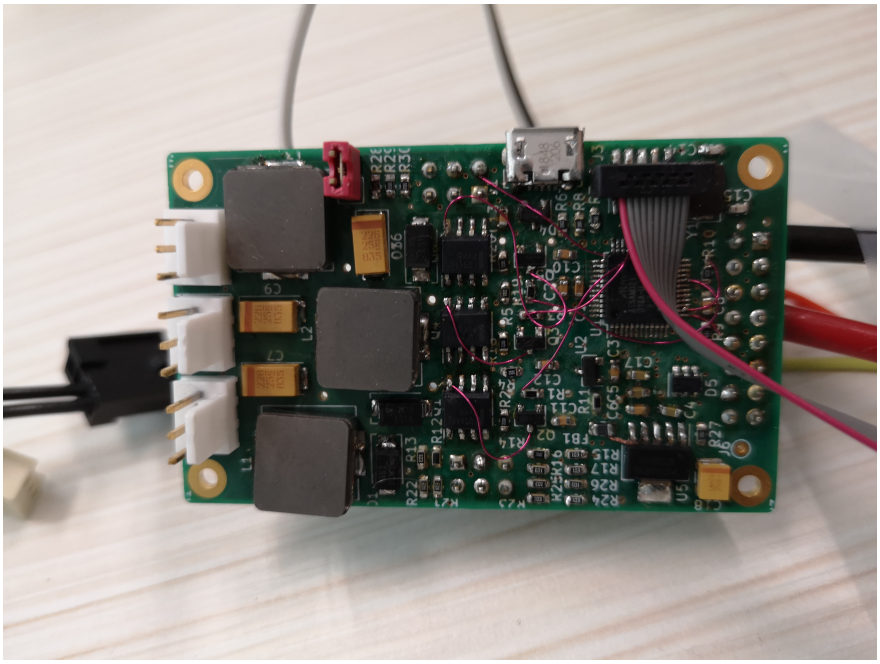

Fig. 2.1: The first MoniMod prototype

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 forces it to only change slowly.

Note: Here, a PMOS device has been used; this will be replaced with a radiation-tested high gate driver and NMOS combination.

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 very simple: it is just resistor dividers. An impedance of 100k 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.

Note: The temperature sensors are currently of the LM61 type, not requiring any biasing, but in the next revision an optionally mounted current source will be added to enable using PT100 / PT1000 sources.

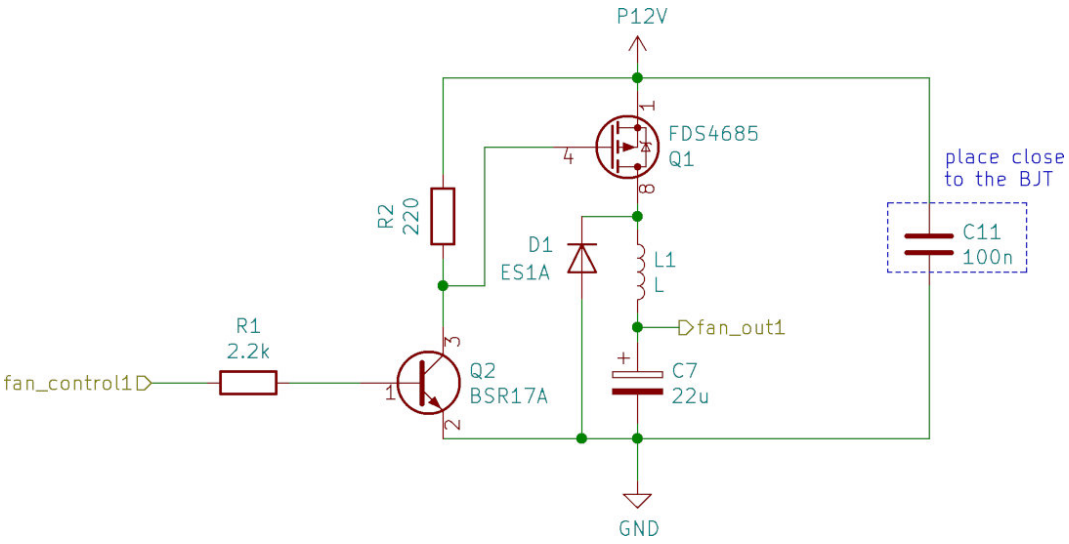

Fig. 2.2: Schematic of the fan driver buck circuit

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.

Table 2.1: J2 connector 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 connector 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 connector pinout

pin	name	pin	name
1	TMP1	2	ADDR0
3	TMP2	4	ADDR1
5	TMP3	6	ADDR2

Table 2.4: J5 connector pinout

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

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 and extended 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*, *cmd_space_t *impl_ext_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 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

uint8_t **user_data**[252]
provide some (optional) user data storage

3.3 PMBus and extended commands overview

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

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

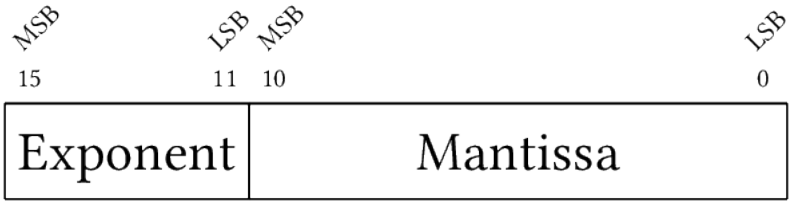

Fig. 3.1: The PMBus Linear data format

Table 3.1: PMBus commands implemented by the MoniMod

Command code	Command name	Transaction type	Data length	Description
00	PAGE	Byte write / read	1	set get page
1A	QUERY	Block w / r proc. call	1	query cmd props
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
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
FF	PMBUS_COMMAND_EXT	Extended command	var	access extended cmds

Table 3.2: Extended commands implemented by the MoniMod

Command code	Command name	Transaction type	Data length	Description
01	WRITTEN_FW_SIZE	Word write	2	size of the FW to be written
02	WRITTEN_FW_BLOCK	MultiByte write	8	FW block to be written
03	WRITTEN_FW_CHKSUM	Word write	2	checksum of the written FW
05	BOOT_NEW_FW	Byte write	1	turn on btldr pgm mode, reset
06	UC_RESET	Byte write	1	reset the uC
A0	TMR_ERROR_CNT	MultiByte read	4	get TMR error count
B0	USE_PEC	Byte write / read	1	turn PEC on / off
C0	TEMP_CURVE_POINTS	MultiByte write / read	13	set / get temp. curve points
C1	TEMP_MATRIX_ROW	MultiByte write / read	7	set / get temp. matrix points
C4	TC_ONOFF	Byte write / read	1	turn temp. control on / off

3.4 Detailed list of PMBus and extended 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 $0 \leq N \leq 2$.

3.4.2 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.

3.4.3 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 3.3. The two bits that set the tachometer pulses / revolution, which take the values 0–3, correspond to 1–4 pulses per revolution.

Table 3.3: 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

3.4.4 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.

3.4.5 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).

3.4.6 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).

3.4.7 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.

3.4.8 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.

3.4.9 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).

3.4.10 MFR_ID

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

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

3.4.11 MFR_MODEL

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

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

3.4.12 MFR_REVISION

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

This returns the manufacturer revision string.

3.4.13 MFR_LOCATION

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

This returns the manufacturer ID string, “Geneva”.

3.4.14 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).

3.4.15 MFR_SERIAL

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

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

3.4.16 PMBUS_COMMAND_EXT

Command code: **FF**
Transaction type: **Extended command**
Data length: **var**

To access the extended commands, described below, the PMBUS_COMMAND_EXT is used: the command code of the extended command is passed as the next data byte and the rest of the transaction continues like a regular command.

3.4.17 (ext.) WRITTEN_FW_SIZE

Ext. Command code: **01**
Transaction type: **Word write**
Data length: **2**

Before writing a new FW binary through the bootloader, its size in bytes has to be given using this command.

3.4.18 (ext.) WRITTEN_FW_BLOCK

Ext. Command code: **02**
Transaction type: **MultiByte write**
Data length: **8**

A new binary is written to the bootloader in consecutive chunks of 8 bytes, using this command.

3.4.19 (ext.) WRITTEN_FW_CHKSUM

Ext. Command code: **03**
Transaction type: **Word write**
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 checked with this command. This command also resets the write pointers.

3.4.20 (ext.) BOOT_NEW_FW

Ext. Command code: **05**
Transaction type: **Byte write**
Data length: **1**

The BOOT_NEW_FW command passes execution to the bootloader. A special code is written to the flash memory to direct the bootloader to switch to PMBus mode, supporting extended commands. When already in bootloader mode, this clears the special code and boots to the main FW, instead.

3.4.21 (ext.) UC_RESET

Ext. Command code: **06**
Transaction type: **Byte write**
Data length: **1**

Writing any byte to this command triggers a uC reset.

3.4.22 (ext.) TMR_ERROR_CNT

Ext. Command code: **A0**
Transaction type: **Word read**
Data length: **4**

When software mitigation through COAST is enabled (see Section 3.8.1), one can access the TMR_ERROR_CNT counter using this command.

3.4.23 (ext.) USE_PEC

Ext. Command code: **B0**
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. Using this command with a zero byte disables PEC; any non-zero value enables it. The command itself is used without a PEC byte appended, no matter whether the function is enabled or not.

3.4.24 (ext.) TEMP_CURVE_POINTS

Ext. Command code: **C0**
Transaction type: **Block write / read**
Data length: **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

3.4.25 (ext.) TEMP_MATRIX_ROW

Ext. Command code: **C1**
Transaction type: **Block write / read**
Data length: **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.

3.4.26 (ext.) TC_ONOFF

Ext. Command code: **C4**
Transaction type: **Byte write / read**

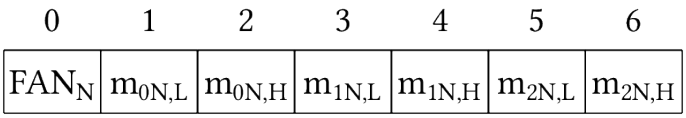

Fig. 3.3: Temperature matrix data frame

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.

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

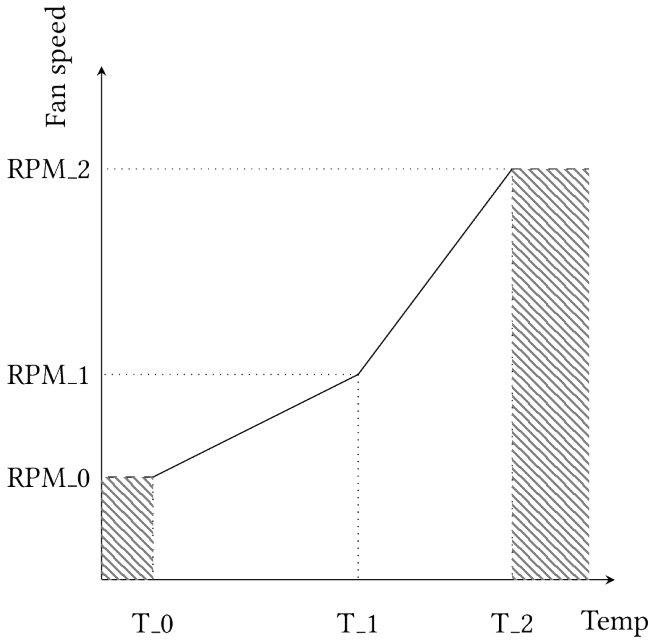

Fig. 3.4: Temperature curve

- 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

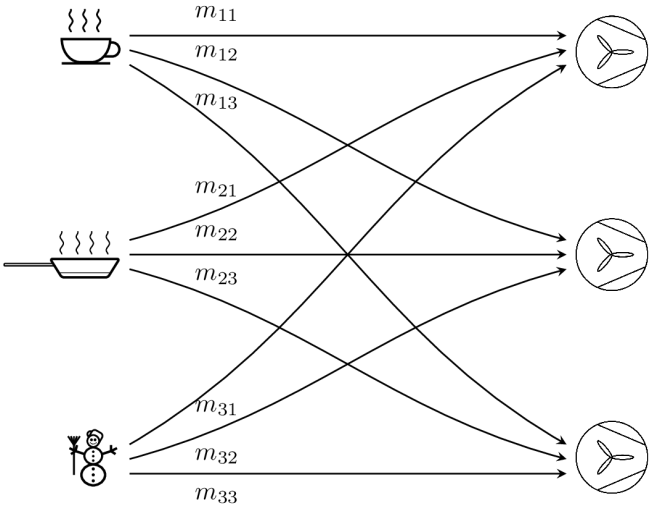

Fig. 3.5: Temperature mixing matrix

3.7 Test firmware

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

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