SAPIENZA
UNIVERSITÀ DI ROMA

AVR Multi Motor Control

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Corso di Laurea in Ingegneria Informatica e Automatica

Candidate
Paolo Lucchesi
ID number 1765134

Thesis Advisors
Prof. Giorgio Grisetti
Drs. Barbara Bazzana

Academic Year 2021/2022

Thesis not yet defended

Reviewer:
Prof. Silvia Bonomi

AVR Multi Motor Control
Bachelor's thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Website: https://github.com/jcondor98/ammc
Author's email: lucchesi.1765134@studenti.uniroma1.it

Dedicated to my family, my granddad Pietro, my mom Anna Rosa, my dad Marco, my grandmom Pierina and my sister Valentina, which I thank everyday for everything I am.
To my dearest friends in Pitigliano, with whom I share some of the most beautiful memories I have.
To Nicola, with whom I have shared part of this path; he helped me in a dark period of my life and he is one of my dearest friends.

Abstract

Nowadays, men physical work is gradually being taken or eased by technology. Many solutions in this area of interest rely on the use of electrical motors to accomplish tasks. Furthermore, electrical motors are widely used in many retail products, such as toys, domotic devices and so on.

In this work I propose, as the result of my bachelor degree apprenticeship, the *AVR Multi Motor Control* ecosystem as a solution for the management of multiple dc motors. It is composed of a *client* application for user interaction, a *master* controller and multiple *slave* controllers, each handling a single motor. The master controller manages slave controllers, reading and manipulating the dc motors' speed with granularity.

In chapter 1 I give a broad overview on the work done, focusing on the structure of the entire project and the approach I had relating to software architecture, coding and documentation writing.

In chapter 2 I discuss the client-side application, focusing on user interaction and software architectural characteristics.

In chapter 3 I discuss the master controller, focusing on hardware setup and firmware internals.

In chapter 4 I discuss the slave controller, focusing on hardware and the software-defined Proportional-Integral-Derivative controller embedded in the firmware.

In chapter 5 I discuss the protocols responsible for the communication between the client application and the master controller, and how I used the underlying serial communication channel.

In chapter 6 I explain how I used the I2C bus to connect the master controller to multiple slave controllers.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

The use of dc motors is widespread in many application fields. Often, the offered solutions are monolithic and use of very specific (and sometimes ad-hoc developed) hardware. The ammc ecosystem focuses on the use of very common, affordable and easily available hardware, such as AVR microcontrollers.

The software is designed to be modular, extensible and easily hackable by any average C programmer, so that in the future other people interested in this project can contribute or adapt ammc to fit their needings and accomplish their business logic.

1.2 Contributions

The ammc ecosystem offers a flexible, affordable, hackable solution for the management of multiple dc motors. It is composed of three main components:

- A client-side application
- A master controller
- One or more slave controllers

1.2.1 Client

The client-side application is written in C. It makes use of the POSIX standard library, and in particular the *termios* interface to generic serial devices. It offers a shell-fashioned terminal user interface.

1.2.2 Master

The master controller is an AVR microcontroller unit. The firmware is written in C, using the *avr-gcc* compiler and the *avr-libc* standard library implementation. It dispatches user commands (taken by the client) to slave controllers.

1.2.3 Slave

The slave controllers are AVR microcontroller units. The firmware is written in C, using the *avr-gcc* compiler and the *avr-libc* standard library implementation. Each slave controller manages a single dc motor, taking commands by the master controller.

1.3 Technical approach

1.3.1 Software architecture

I have developed all the top-level ammc components following the principles I have learned both from my University courses[3] and my personal insights[5][7].

The client application and the microcontroller firmwares were developed following the *SOLID* principles, with attention on the *Open-Close principle*. I have put particular attention on modularity; many of the software modules found in all the ammc applications are standalone and will work out of the box if brought into other projects.

I tried to develop my software with the following policy: the core of the application should be composed of reusable, standalone modules; the business logic of the application, which obviously changes radically from project to project, should be placed in the peripheral modules and should use, when possible, interfaces for dependency inversion towards the core modules.

For example, in the master controller firmware source code I use function pointers to generalize the operations to be triggered by client commands. Naturally, such practices would have been easier and more evident with a programming language explicitly supporting the object-oriented paradigm (e.g. C++).

1.3.2 Documentation

Documentation is paramount in any software project; Damian Conway once said: ”Documentation is a love letter that you write to your future self”. I did my best to write precise and exhaustive documentation for every software piece I realized.

This thesis is by itself a document dealing with every aspect of the ammc project. The `README.md` file in the root directory of the repository offers a more concise resume.

All the source code (the head files in particular) are exhaustively documented with *doxygen*, a tool which automatically converts properly formatted comments found in the code into technical documentation; such resource is available in the `gh-pages` git development branch and is hosted using GitHub Pages.

The client-side application is documented with a Unix man page, which can also be found in this thesis in section 2.5.

Chapter 2

Client-side user interaction

A client program has been realized to manipulate the dc motors directly from the PC. It can get and set the speed of individual motors, and to apply all the previously set speeds for all of them at once.
A brief list of the client's features is given below:

- Granular handling for getting and setting motors' speed
- Modular and extensible software architecture
- Terminal User Interface, implemented as a command shell
- Support for non-interactive use (i.e. scripting)
- Communication with master controller using the serial protocol
- Compatible with POSIX-compliant environments

The client is also documented with a man page, which can be found in section 2.5.

2.1 User Interface

The end user interacts with the whole ammc ecosystem using a text-based client. It consists in a shell module, which I had written myself, offering some *internal commands* (hardcoded in the shell module itself) and is extended by *external commands* (found in a separated source code entity, and that can even be compiled in a detached transaction unit).

A particular focus was made on the software architecture: indeed, every external command can be realized standalone, and it is easy to add new commands just by altering the `client/source/shell_commands.c` source file.

2.2 Primitives offered

The commands that can be used to interface with the ammc ecosystem are the following:

connect <device-path> Connect to a master controller, given the path to the block device representing it.

get-speed <motor-id> Get the speed of a dc motor given its id. The motor id must be specified as a decimal number.

set-speed <motor-id>=<speed> Set the speed of a dc motor given its id. The motor id must be specified as a decimal number and the speed must be specified in rpm.

apply Apply the previously set speed for all the dc motors.

set-slave-addr Set a new TWI address for a slave controller.

2.2.1 Non-interactive mode

The client shell is capable of running in non-interactive (i.e. scripting) mode with the `-s` option. If so, it will parse the input from a specified text file, or from `stdin` if not provided. A shell launched in non-interactive mode will not print shell prompts, and exit when end-of-file is encountered or on command failure.

2.3 Serial module

The client's serial module has been realized using the POSIX *termios* interface. Unlike the master controller's counterpart, all its code is reentrant, therefore multiple instances of multiple serial devices can theoretically exist at the same time.

From the client's perspective, the master controller is seen as a file descriptor, and the end user just have to specify the path of the block device file representing the serial communication channel (e.g. `/dev/ttyACM0`) using the `connect` command.

2.4 Specification

2.4.1 Software modules

An exhaustive list of software modules for the client application is given in table 2.1. File paths are relative to the `client/` directory.

2.4.2 Modules dependency graph

The dependency graph for the client application software modules is shown in figure 2.1. The *debug* module is excluded. Notice how the dependency flow goes in one direction (from top to bottom); this is the result of the software architecture policy used.

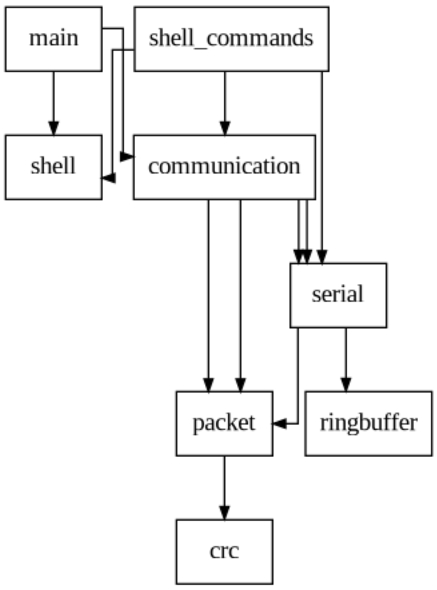

Figure 2.1. Client application modules dependency graph

2.5 Man page

NAME

ammc — AVR Temperature Monitor client

SYNOPSIS

ammc [**-s** [*script*]] [**-c** *device-file*]
ammc -h

DESCRIPTION

ammc is a multi-motor control device realized with an AVR microcontroller; this program is an ad-hoc, tui client to interface it. The ammc device is designed to work with DC motors with an embedded or external encoder.

Options

-c *device-file* Connect to the device identified by *device-file* (e.g. /dev/ttyACM0)

-s [*script*] Execute in script mode (do not print prompt, exit on error etc...), being *script* a file containing a command each line. If *script* is not given, standard input is used

-h Display a help message and exit

Commands

help [*command*] Show help, also for a specific command if an argument is given

connect <*device_path*> Connect to an ammc Master MCU given its device file (usually under /dev)

disconnect Close an existing connection - Has no effect on the device

dev-echo <*arg*> [*arg2 arg3 ...*] Send a string to the device, which should send it back

get-speed <*motor-id*> Get the speed of a DC motor in RPM

set-speed <*motor-id*>=<*value*> Set the speed of a DC motor in RPM

AUTHOR

Paolo Lucchesi <paololucchesi@protonmail.com>
https://www.github.com/jcondor98/ammc

Module	Description	Files
communication	Contains all the top-level communication routines	`include/communication.h`, `source/communication.c`
crc	Contains the CRC generation and checking routines	`include/crc.h`, `source/crc.c`
debug	Contains convenient debug facilities	`include/debug.h`
main	Contains the main client application routine	`source/main.c`
packet	Contains packet generation and manipulation routines	`include/packet.h`, `source/packet.c`
ringbuffer	Circular buffer implementation for the serial module	`include/ringbuffer.h`, `source/ringbuffer.c`
serial	Contains all the routines for the underlying serial communication layer	`include/serial.h`, `source/serial.c`
shell	Main program shell with built-in commands	`include/shell.h`, `source/shell.c`
shell_commands	Contains the custom external commands to interact with the master controller	`source/shell_commands.c`

Table 2.1. Client application software modules

Chapter 3

Master controller

The master controller handles all the slave controllers, dispaching arbitrary commands to them using the I2C protocol. It also communicates directly with the client application via serial port.

3.1 Hardware setup

The master controller itself is an AVR *ATMega2560* microcontroller unit[1]. This particular MCU has features convenient for this project, such as:

- I2C dedicated hardware subsystem
- Serial-over-USB bridge
- Relatively powerful specifications for future feature adding
- Plenty of timers and outgoing power pins

A $100pF$ capacitor is used to block the reset capabilities of the serial-over-usb controller and enhance the serial channel reliability. Two $4.7k\Omega$ resistors are used as open-drain resistors for the I2C bus.

3.2 I2C setup

The I2C protocol is used to communicate with slave controllers. Transmission and reception are interrupt-based, and no busy-wait is used.

The I2C module has broadcasting capabilities, according to the informations found in the I2C standard I2C standard[6]; the broadcasting (i.e. *general call*) address used is `0x00`. As stated by the standard, the master controller has no knowledge on the number or identity of the slaves receiving a broadcast frame.

3.3 Power management

An own-written wrapper for the avr-gcc standard library's sleep functionalities is used for power management. By default, the master controller is in *idle* mode, and it is awakened by any raised interrupt, e.g. by incoming serial or I2C data; then, its main loop routine is executed and, if no other operations must be performed, it returns in idle mode.

The master controller is also put in idle when waiting for data inside serial or I2C routines; this is possible thanks to the interrupt-driven nature of the aforementioned modules.

3.4 Specification

3.4.1 Software modules

An exhaustive list of software modules for the master controller firmware is given in table 3.1. File paths are relative to the `master/` directory.

3.4.2 Modules dependency graph

The dependency graph for the master controller's software modules is shown in figure 3.1. The *debug* module is excluded. Notice how the dependency flow goes in one direction (from top to bottom); this is the result of the software architecture policy used.

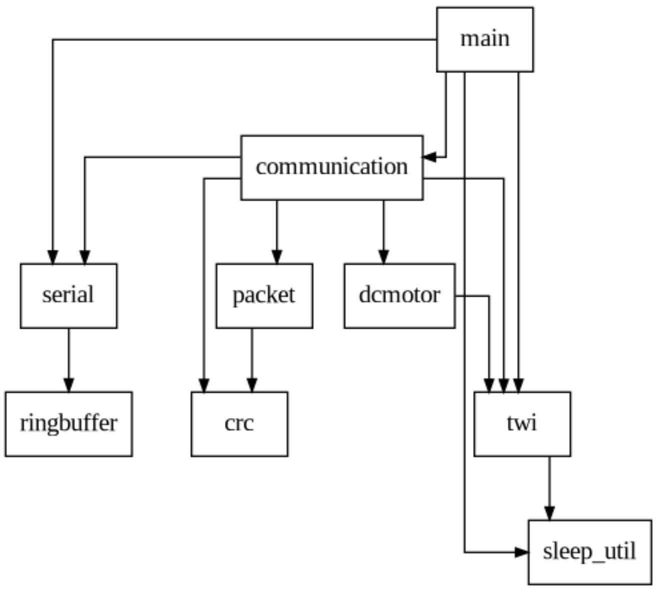

Figure 3.1. Master controller modules dependency graph

3.4.3 Circuit schematics

The complete circuit schematics for the master controller and all its components is shown in figure 3.2.

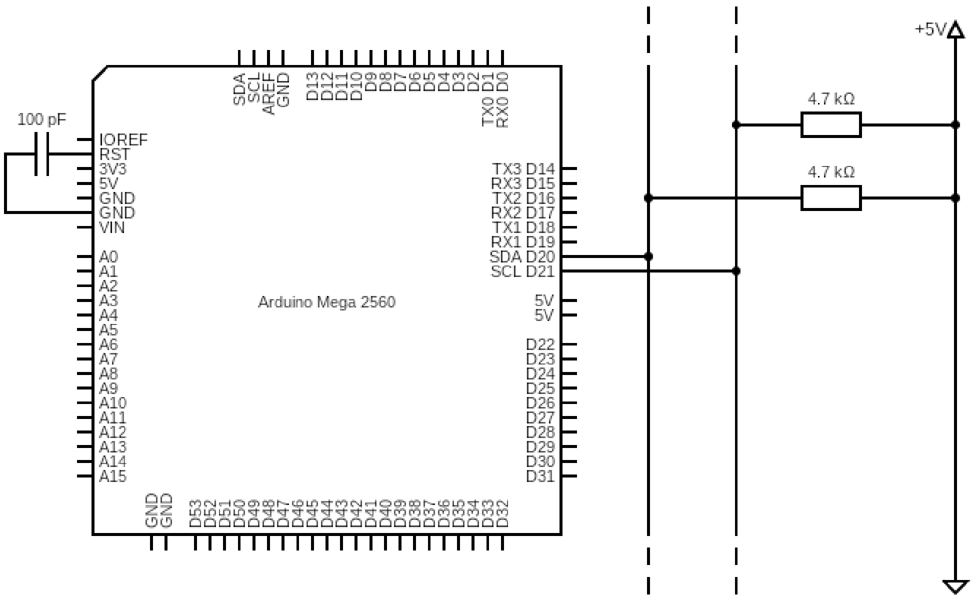

Figure 3.2. Master controller schematics

3.4.4 Wiring

The wiring correspondence for the master controller MCU is shown in table 3.2.

Module	Description	Files
communication	Contains all the top-level communication routines	`include/communication.h`, `source/communication.c`
crc	Contains the CRC generation and checking routines	`include/crc.h`, `source/crc.c`
dcmotor	Contains the top-level routines for interfacing slave controllers	`include/dcmotor.h`, `source/dcmotor.c`
main	Contains the main and power management routines	`source/main.c`
packet	Contains packet generation and manipulation routines	`include/packet.h`, `source/packet.c`
ringbuffer	Circular buffer implementation for the serial module	`include/ringbuffer.h`, `source/ringbuffer.c`
serial	Contains all the routines for the underlying serial communication layer	`include/serial.h`, `source/serial.c`
sleep_util	Handy wrapper for the avr-libc power management facilities	`include/sleep_util.h`
twi	Contains the I2C/TWI layer which underlies the communication between master and slaves	`include/twi.h`, `source/twi.c`

Table 3.1. Master application software modules

Board pin	MCU pin	Description
D20	44/PD1/INT1/SDA	I2C data bus
D21	43/PD0/INT0/SCL	I2C clock bus
RESET	RESET	On-board MCU reset pin

Table 3.2. Master controller wiring

Chapter 4

Slave controllers

Each slave controller directly handles a single dc motor, receiving commands from the master controller via I2C. It must offer an interface to get and manipulate its motor's speed.

The slave controller produces a PWM wave having a duty cycle consistent with the speed (given in RPM) specified by the end user in order to control its motor.

Furthermore, it comes with a software Proportional-Integral-Derivative controller to correct the bias between target and actual motor speeds.

4.1 Hardware setup

The slave controller itself is an AVR *ATMega328P* microcontroller unit[2], which offers the following features of interest:

- CH340 Serial-over-USB controller for convenient firmware flashing
- Reduced size
- Relatively high clock and resources
- Dedicated I2C hardware subsystem
- PWM-capable 8-bit and 16-bit timers

Apart from the motor itself (along with its power supply and eventually its dedicated control board) no additional hardware is used.

4.2 I2C setup

A slave controller communicates with the master controller using the I2C protocol. As for the master controller, both transmission and reception are interrupt-based.

The slave controller is always passive on the bus, waiting to be addressed by the master.

4.2.1 Dynamic I2C addressing

Each slave controller needs a unique I2C address. Basically, there are two ways to assign an arbitrary address to a slave.

The first and most trivial one is to define the `TW_DEFAULT_ADDR` macro to the desired address when building the slave controller firmware. This solution is impractical for obvious reasons and should be avoided.

The second (and suggested) method is to connect the master and slave controllers via I2C and the master controller with the client via serial. Then the `set-slave-addr` command can be used to change the I2C address to the desired one. The changes take place immediately, as the slave controller's I2C software module is reinitialized, and they will be saved in the EEPROM embedded into the AVR microcontroller for durability.

4.3 Proportional-Integral-Derivative controller

An own-written, software-defined Proportional-Integral-Derivative controller is used to correct the actual motor speed. As for any PID device, the equation controlling the error is:

$$u(t) = K_p e(t) + K_i \int_{t_0}^{t} e(\tau)\, d\tau + K_d \frac{d}{dt}\, e(t) \tag{4.1}$$

being:

- $u(t)$ the PID control variable
- K_p the proportional gain
- K_i the integral gain
- K_d the derivative gain
- $e(t)$ the measured speed error

4.3.1 Measuring speed

For actual PID capabilities, the slave controller must sample the actual speed of the dc motor at fixed intervals. Therefore, the motor must have an embedded, two-phase digital encoder.

In fact, one phase is wired to an interrupt-enabled input pin, so the slave controller is notified immediately when an encoder signal (raising edge) is generated. Every sampling intervals, the PID routine takes the motor's actual position (measured by the cumulative number of encoder triggers) and computes its real speed.

4.3.2 Approximations

Of course, a software digital PID controller can not compute exact integrals and derivatives, so it is necessary to use approximation.

The integral operation is approximated using *Riemann sums*. Given a time delta Δt (the actual speed sampling interval in this particular case), the approximating law is defined as:

$$\int_{t_0}^{t} e(\tau)\, d\tau \approx \sum_{k=1}^{n} e(t_k^*) \cdot \Delta t \tag{4.2}$$

The derivative operation is approximated using its definition[4]:

$$\frac{d}{dt}\, e(t) = \lim_{\Delta t \to 0} \frac{e(t+\Delta t) - e(t)}{\Delta t} \implies \frac{d}{dt}\, e(t) \approx \frac{e(t+\Delta t) - e(t)}{\Delta t} \tag{4.3}$$

being Δt a time delta (again, the actual speed sampling interval).

4.4 Power management

The same avr-libc wrapper written for the master controller power management is used. By default, the slave controller is in *idle* mode, and it is awakened by any raised interrupt. The main loop routine is executed and the idle mode is entered again. Like for the master controller, the slave is put in idle when waiting for the I2C routines to return.

The AVR idle mode allows the timers to work, so the slave controller can generate PWM waves to control the motors, even while sleeping.

4.5 Specification

4.5.1 Software modules

An exhaustive list of software modules for the slave controllers firmware is given in table 4.1. File paths are relative to the `slave/` directory.

4.5.2 Modules dependency graph

The dependency graph for the slave controllers' software modules is shown in figure 4.1. The *debug* module is excluded.

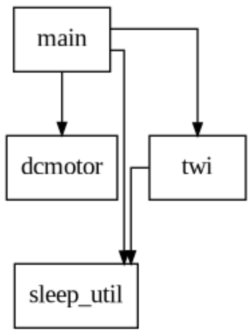
$$\begin{tikzcd}
 & \text{main} \arrow[dl] \arrow[dr] \arrow[d] & \\
\text{dcmotor} & \text{} & \text{twi} \arrow[dl] \\
 & \text{sleep\_util} &
\end{tikzcd}$$

Figure 4.1. Slave controller modules dependency graph

4.5.3 Circuit schematics

The complete circuit schematics for the slave controller and all its components is shown in figure 4.2. This circuit can be repeated multiple times (up to 126) in a single ammc ecosystem.

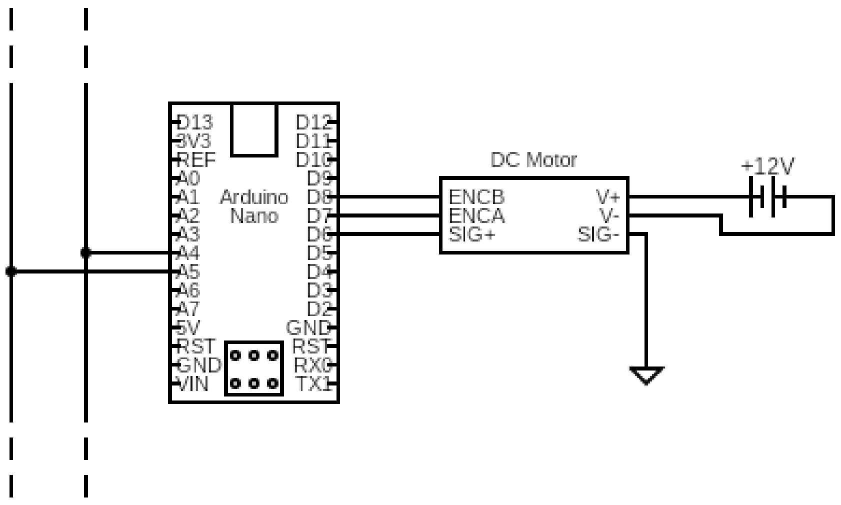

Figure 4.2. Slave controller schematics

4.5.4 Wiring

The wiring correspondence for the slave controller MCU is shown in table 4.2.

Module	Description	Files
dcmotor	Contains stuff to directly manipulate dc motors, including the PID controller, PWM waves generation and speed manipulation routines	`include/dcmotor.h`, `source/dcmotor.c`
main	Contains the main and power management routines	`source/main.c`
sleep_util	Handy wrapper for the avr-libc power management facilities	`include/sleep_util.h`
twi	Contains the I2C/TWI layer which underlies the communication between master and slaves	`include/twi.h`, `source/twi.c`

Table 4.1. Slave application software modules

Board pin	MCU pin	Description
A4	27/PC4/ADC4/SDA	I2C data bus
A5	28/PC5/ADC5/SCL	I2C clock bus
D6	10/PD6/AIN0/OC0A	PWM output to the dc motor
D7	11/PD7/AIN1/ICP1	Encoder phase A, interrupt-enabled
D8	12/PB0/CLK0	Encoder phase B

Table 4.2. Slave controller wiring

Chapter 5

Client-Master communication

The client application and the master controller communicate over a protocol built on top of the serial-over-usb layer. Such protocol is completely binary and packet-based. Each packet has a variable length (with a total maximum size of 36 bytes) and its integrity is checked with a trailing CRC-8 checksum.

The protocol comes with a simple handshaking mechanism, in order to synchronize the packet IDs between endpoints and as a shallow proof of correct functionality of the communication layers.

5.1 Serial layer

Data exchange between client and master relies on the serial protocol. For the client application, the *termios* library is used, while for the master controller the (hardware) serial subsystem offered by the AVR microcontroller[1] is used.

The serial communication is set up as follows:

- Baud rate of 115200 baud, double speed transmission
- 8 bit frame width
- No start bits, 1 stop bit (i.e. 8N1)
- No embedded parity bit

For the master controller, both transmission and reception are interrupt-driven, so the communication main routine is only called when data is actually available.

5.2 Packet headers

Each packet's metadata is contained in a fixed size header, composed as describe in table 5.1.

5.2.1 Identifiers

The *id* field stores the packet identifier, which is incremental and can be repeated in a single communication session. When the handshake is performed, or when any endpoint raises an error (issuing a *NAK* packet), the id is reset to zero for both sides. Each *ACK* and *NAK* packet is generated with the same id of the referred packet (see 5.3 for further informations).

5.2.2 Packet types

The *type* field stores the packet type. An exhaustive list of packet types is given in table 5.2

5.2.3 Motor selector

The *selector* field is used to store the identifier for the dc motor to manipulate. This field is used in `GET_SPEED` and `SET_SPEED` packets.

For *NAK* packets, instead, the *selector* field is used to store error codes (found in table 5.3).

For all the other packet types, the *selector* field is ignored.

5.3 Acknowledgements and errors

A communication endpoint must wait for an acknowledgement message from the counterpart once it sent a packet in order to send a new one. ACK and NAK packets do not bring any data.

When a packet arrives, it is checked for integrity and sanity. If it is sane, then an ACK packet is sent; if not, then a NAK packet is sent. ACK and NAK packets are simply discarded if corrupted in some way.

A NAK packet uses the *selector* field to send to the other endpoint the error code describing what happened on its side. Error codes are listed in table 5.3.

Field	Size (bits)	Description
id	8	Packet ID
type	8	Packet type
selector	8	Selector for DC motors. Also used to store error codes in NAK packets
size	8	Total packet size, including header and checksum

Table 5.1. Packet header fields

Type	Code	Description
`NULL`	`0x00`	Reserved, never use
`HND`	`0x01`	Handshake
`ACK`	`0x02`	Acknowledgement
`NAK`	`0x03`	Communication error
`ECHO`	`0x04`	Echo between client and master (debug only)
`TWI_ECHO`	`0x05`	Echo a single char to the first slave via TWI (debug only)
`GET_SPEED`	`0x06`	Get the current speed for a dc motor
`SET_SPEED`	`0x07`	Set (and apply) the speed for a dc motor
`APPLY`	`0x08`	Tell all the slaves to apply the previously set speeds
`DAT`	`0x09`	Primarily used for responses from the AVR device
`SET_ADDR`	`0x0A`	Change the I2C address of a slave
`LIMIT`	`0x0B`	Used for sanity checks - Must have highest value

Table 5.2. Exhaustive list of packet types

Error	Code	Description
`SUCCESS`	`0x00`	No errors encountered
`ID_MISMATCH`	`0x01`	Id of received packet is not consistent
`CORRUPTED_CHECKSUM`	`0x02`	Checksum mismatch, received packet is corrupted
`WRONG_TYPE`	`0x03`	Received packet has invalid type
`TOO_BIG`	`0x04`	Received packet has invalid size (too big)

Table 5.3. Exhaustive list of packet types

Chapter 6

Master-Slave communication

The master and slave controllers communicate to each other using the I2C bus. Naturally, the master controller is responsible for starting communication sessions, addressing one or all the slaves.

The bitrate is set to 100kbit/s, i.e. the maximum speed available for the original I2C standard[6] (excluding fast modes, which were implemented later).

6.0.1 Hardware setup

The I2C bus itself is a pair of bus rails, named `SDA` and `SCL`, representing data and bus clock respectively.

Two $4.7k\Omega$ resistors connects `SDA` and `SCL` to a $5V$ voltage generator and act as *open-drain* resistors.

6.0.2 Communication frames

The first byte of the I2C communication frame (excluding, of course, the slave address) is an 8-bit unsigned integer representing the command sent by the master to the slave. The trailing data represents an argument, with its size and shape depending on the type of command issued. Command codes, with their specific arguments, are shown in table 6.1.

Command	Code	Arg. size (bytes)	Description
`DC_MOTOR_CMD_GET`	`0x00`	-	Get the dc motor speed in rpm
`DC_MOTOR_CMD_SET`	`0x01`	1	Set the dc motor target speed
`DC_MOTOR_CMD_APPLY`	`0x02`	-	Apply the previously set target speed
`TWI_CMD_ECHO`	`0x03`	1	Send back the received character to the master (debug)
`TWI_CMD_SET_ADDR`	`0x04`	1	Set a different I2C slave address

Table 6.1. Master-to-slave commands

Chapter 7

Conclusions

In this work I developed an electrical motor handling hardware/software solution with modularity and extensibility in mind. In this very final chapter I demonstrate how ammc can be used in practice and I explain how it could be developed further in the future.

7.1 Example setup

7.1.1 Hardware

The hardware required for the example setup given here is the following:

- One *ATMega2560* microcontroller unit for the master controller
- One *ATMega328P* microcontroller unit for the slave controller
- One dc motor with embedded two-phase encoder
- Two $4.7k\Omega$ resistors for the I2C open-drain
- One capacitor of about $100pF$ to inhibit the reset-over-serial capabilities of the ATMega2560 board

Optionally, a dedicated motor board may be used.

The hardware setup must be performed as shown in figures 3.2 and 4.2. The motor must be powered consistently with its requirements (usually $12V$). The AVR microcontrollers can be powered using their USB sockets. The I2C reference voltage can be set by using a $5V$ output pin of the master controller board.

7.1.2 Software

The software can be built and eventually flashed using GNU Make. the *avr-gcc* toolchain and the *avr-libc* library are required for the build, while *avrdude* is required for flashing.

The `make` command will build all the ammc software, i.e. the master and slave controllers firmwares and the client application. The `make master-flash` and the `make slave-flash` commands will flash the master and slave firmwares into their respective microcontrollers.

At last, the user may want to use the `set-slave-addr` command to assign an arbitrary address to the only slave controller.

7.2 Future work

7.2.1 Extended connectivity

The extension of the master controller connectivity would make ammc suitable for whole new range of application possibilities. For example, *LoRaWAN* and similar communication vectors would make ammc a motor management system suitable for dc motor distributed in a wide-area. The use of multiple wired low-level communication means such as SPI or raw serial ports would make ammc suitable for a multi-master, multi-group setup.

7.2.2 Programming API

The client application offers a non-interactive mode for basic scriping functionality. Thus it is good for basic automation, it is quite limited when more complex features are required, such as an advanced use of logic or control flow capabilities. For this purpose, the implementation of a programming API, eventually with bindings in various languages, would make the ammc functionality directly available to the programmer as high-level functions.

Bibliography

[1] Atmel. *Atmel ATmega640/V-1280/V-1281/V-2560/V-2561/V datasheet*, 2014.

[2] Atmel. *Atmel ATmega328P datasheet*, 2015.

[3] Giuseppe De Giacomo. Corso di Fondamenti di Informatica 2 - Progettazione del Software. `https://sites.google.com/a/diag.uniroma1.it/fondamenti-di-informatica-ii-progettazione-del-software-2017-18/`, 2018.

[4] Doron Levy. Introduction to Numerical Analysis. `http://math.umd.edu/~dlevy/classes/amsc460/na-notes.pdf`, 2017.

[5] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Addison-Wesley, 2017.

[6] Philips Semiconductors. *The I2C-bus Specification*, 2000.

[7] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 4th edition, 2013.