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

Contents

Chapter 1

Introduction

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.

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

2.5 Man page

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[4]; 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.

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
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

3.4.2 Modules dependency graph

3.4.3 Circuit schematics

3.4.4 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.

4.3 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[3]:

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

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.

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

4.5.2 Modules dependency graph

4.5.3 Circuit schematics

4.5.4 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 TODO 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
LIMIT	0x0A	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

Chapter 7

Conclusions

Bibliography

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

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

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

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