NTNU
Innovation and Creativity

CyberBike

Audun Sølvberg

Master of Science in Engineering Cybernetics
Submission date: June 2007
Supervisor: Amund Skavhaug, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem Description

The Department of Engineering Cybernetics at NTNU wish to complete a project on an autonomous unmanned bicycle - the ``CyberBike'' - which have been subject to development by the department's students during the past few years.

Work that has to be done to reach this goal involves:
- software development with real-time requirements
- use of real-time operating systems
- electronic/hardware development and circuit analysis
- control theory
- mechanical engineering
- power electronics

The candidate have to i.a.:
- gain an understanding of the existing system
- on an independent basis, point out and suggest what has to be done
- as far as time permits; implement the suggested solutions

Assignment given: 08. January 2007
Supervisor: Amund Skavhaug, ITK

CyberBike

Master's thesis

Audun Sølvberg
audunsol@stud.ntnu.no

Engineering Cybernetics
Industrial Computing

Supervisor:
Amund Skavhaug

Hand in date: June 8, 2007

Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Engineering Cybernetics
TRONDHEIM

Preface

Now my leg of the CyberBike relay race is over. The development process of ITK's autonomous unmanned bicycle however, still has to run some more laps. But the last semester it certainly have been approaching its goal, hopefully close enough for the next student working on it to get the CyberBike up and running.

Some valuable personal experiences has been gained, during my work on this project. One of the most important is the lowering of my own threshold for working with hardware. By the start of this thesis, etching my own circuit board was nothing I'd ever expect to do.

Furthermore; during the work on this project, I have been facing a lot of hardware problems. This was clearly not desirable, but at the end some important lessons around debugging have been learned. Isolating an error in order to point out where it resides, should be a self-explanatory approach to every debugging process. Still; the true value of it has never looked as clear to me as after the work on this thesis.

Here I would like to thank Amund Skavhaug, not just for being an inspiring and flexible supervisor through my past year here at NTNU, but also for being an encouraging and friendly person, giving me challenging and interesting tasks.

My next acknowledgement goes to Øyvind Bjørnson-Langen, my office neighbour, for spending a lot of time on giving advice, lending me his soldering tools, and giving a hand when I needed assistance. Hans Jørgen Berntsen and Terje Haugen at the ITK workshop also deserves an acknowledgement for being oblige and helpful on the tasks involving part and device mounting. Mikael K. Eriksen shared his knowledge and PCB layout on the GPS part, and the "Eurobot guys"; Gunnar Kjemphol and Kristian M. Knausgård spared an DMM-32-AT I/O card and hard disk drive for the CyberBike. It really helped me getting further on the project.

And at last, but not least, I would like to thank my girlfriend Marte for keeping the faith, even though the amount of my time and mind used on this project at times have gotten out of hand.

Trondheim, June 2007

Audun Sølvberg

Contents

Abbreviations and acronyms

A/D	Analog-to-Digital
AC	Alternating current
ADC	Analog-to-Digital Converter
ATA	Advance Technology Attachment Many synonyms and near-synonyms for ATA exist, including abbreviations such as IDE and ATAPI. Also, with the market introduction of Serial ATA in 2003, the original ATA was retroactively renamed Parallel ATA (PATA).
AUAV	Autonomus Unmanned Aerial Vehicle
Bash	Bourne-Again SHell
BIOS	Basic Input/Output System
CD	Compact Disk
CDROM	Compact Disk Read Only Memory
CF	Compact Flash
CPU	Central Processing Unit Often just reffered to as the *Processor*
D/A	Digital-to-Analog
DAC	Digital-to-Analog Converter
DC	Direct Current
DDK	Driver Development Kit
DSP	Digital Signal Processor
EEPROM	Electrically Erasable Programmable Read-Only Memory
EIDE	Enhanced IDE (Integrated Drive Electronics, see *ATA*)

FIFO	First In - First Out
FPGA	Field-programmable gate array (type of PLD)
GB	Giga Byte = 1024 Mega Bytes (MB)
GPS	Global Positioning System
GUI	Graphical User Interface
iC	integrated circuit
I/O	Input/Output
IOV	I/O vector
IDE	Integrated Development Environment
IMU	Inertial Measurement Unit
ITK	Department of Engineering Cybernetics (acronym formed by the Norwegian name; *"Institutt for Teknisk Kybernetikk"*)
ksh	Korn SHell
LED	Ligth Emitting Diode
LOS	Line Of Sight
LSB	Least Significant Bit
LQG	Linear quadratic Gaussian (control)
MB	Mega Byte = 1024 Kilo Bytes (KB)
MHz	Mega Hertz
NC	Normally Closed
NTNU	Norwegian University of Science and Technology (the acronym is formed by the Norwegian name of the university; *"Norges teknisk-naturvitenskapelige universitet"*).
OCB	Open Context Block
OS	Operating System
PCB	Printed Circuit Board
PID	Proportional-Integral-Derivative (controller)

PLD	Programmable Logic Device
PM	Process Manager
POSIX	Portable Operating System Interface The 'X' denotes the Unix (or Unix-like) operating system origin of POSIX
QNX	An RTOS developed by QSSL Was initially called *"QUNIX"* from *"Quantum UNIX"*
QSSL	QNX Software Systems Limited, initially known as *Quantum Software Systems*, see QNX
RAM	Random Access Memory
RM	Resource Manager also known as "device drivers" and "I/O managers"
RTOS	Real-Time Operating System
SBC	Single-board computer
SO-DIMM	Small Outline Dual In-line Memory Module
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus
μC	microcontroller

List of Figures

[1] Single-board computer

List of Tables

List of Code samples

List of Printouts

Abstract

The idea about the CyberBike came to Jens G. Balchen – the founder of the Department of Engineering Cybernetics (ITK) at NTNU – in the 1980's. He wanted to make an unmanned autonomus bicycle, i.e. a bike that could run by itself. The idea was picked up by Amund Skavhaug, who started the CyberBike project in the late 80's. After being deffered for some years, the CyberBike has again gained some attention. This master's thesis is based on Hans Olav Loftum's and Lasse Bjermeland's theses at the spring 2006 and the autumn project of John A. Fossum the same year.

The goal of the CyberBike project is to make the bike work as intended, i.e. as an autonomous unmanned bicycle. This thesis naturally share this goal, although the bike did not become able to take its first autonomous trip within the thesis' time frame.

At the start of the work, the bike were already equipped with a suitcase of computational hardware on its baggage rack, a small QNX Neutrino OS image was installed on the industrial PC mounted in the suitcase, and drivers for the installed motors, tachometers and potmeters were written. An *Inertial Measurement Unit* (IMU) was intended to supply the control system with the necessary information about rotation, acceleration and position, and the unit was purchased for the purpose. Also a driver was written, but not properly tested. The IMU had to be installed and connected to the control system. The bike's control theory was developed, but had never been put into action outside computer simulations (due to the lack of acceleration measurements).

The various tasks that had to be addressed emerged as the development process advanced. First, the IMU had to be connected to the system, by making a signal tranceiver circuit. A small printed circuit board was designed and laid out, mainly to include a MAX233CPP iC[2]. Then the DB-9 serial connector on the bikes single board computer (Wafer-9371A) could be used to read the UART signal from the IMU as RS-232. Then some testing had to be done, and drivers updated.

A better and more advanced IMU (referred to as the "MTi") was added to the project. This unit needed no signal converting circuitry, but driver development and testing still had to be done.

To enhance the CyberBike's navigation opportunities, a GPS module was purchased. A signal transceiving circuit, similar to the one for the IMU, was made for this unit, as well as software to read out the measurements from the device. By the end of this thesis, no navigational algorithms are made, hence the GPS is currently not used, but available for future efforts made on this area.

Some hardware related tasks was carried out, as connecting and implementing functionality to the pendulum limit switches, installation of a emergency stop switch and a power switch, purchasing and installation of two 12V batteries and a cooling fan. An operating system upgrade resulted in replacing the CyberBikePC's storage device, a

[2]integrated circuit

compact flash card, with a mobile hard disk drive. Installation of a motor, for supplying torque to the rear wheel, included setup and tuning of a hardware based velocity controller in a Baldor TFM 060-06-01-3 servo module. However; this task is not to be considered as accomplished, due to some unsolved problems on the system I/O-card's output channels giving the motor controller card its reference voltage.

A bike model and controller realized in Simulink was made by Bjermeland. Hence communicaton between Simulink and the device drivers had to be established, and this was realized by using S-functions and Real-Time Workshop. Finally the controller could be connected to the actual bike, but there was too little time left to explore this thoroughly, and make the system work properly. However, a foundation is laid for further development of the control strategy, hopefully storing a bright future for the CyberBike.

Chapter 1

Introduction

1.1 Motivation

The work on the CyberBike started back in the 1980's, when Amund Skavhaug picked up Jens G. Balchen's idea of developing an autonomously running bicycle. The last few years the CyberBike have again got some attention. The intention of the work behind this thesis is to complete the work of the CyberBike, and take it to its first autonomous outdoor ride.

The Department of Engineering Cybernetics' intentions is to use the bike to get publicity at stands and guided tours at the department. The development of the theory behind the bike's dynamics is also an interesting subject, owing to the fact that none – as far as the candidate of this thesis know – has ever implemented an autonomous bicycle this way, i.e. by controlling an inverted pendulum, the steering angle, and the propulsion speed.

1.2 Problem

The goal of this work is to get the bike running. To do this the instrumentation system has to be completed. A lot have been done at this point, but a working measuring unit for acceleration and rotation have to be included in the system. This was the last impediment for the predecessors of this thesis (see Section 1.3) before the control system could be tested on the bike. From there some modifications and improvements of the CyberBike and its control system are the main subjects of this work.

Some tasks are related to allowing the bike to move without cables attached. A wireless ethernet card is supposed to replace the network cable between a stationary QNX[1] workstation – the host computer – which by the acquisition time of this project is performing most of the network activities with the CyberBikePC via Qnet. Installing batteries to eliminate the other cables, is also a part of the plan.

[1] An RTOS developed by QSSL

1.3 Previous work

The most important previous work to be mentioned here is the master's theses of Loftum [2006] and Bjermeland [2006]. In [Loftum, 2006], basically the development of the bikes instrumentation system is described, with installation of an industrial PC with QNX and creating drivers for the sensors and actuators. Bjermeland [2006] is dealing with the dynamics and the mathematical aspects of the control system. Fossum [2006] is describing some improvements and testing, basically focusing on the work of Loftum [2006].

For a further description of previous work on bicycle dynamics and modeling, refer to [Bjermeland, 2006, chap. 2].

1.4 A comment on outline and language

A comment on the outline of this thesis, and the language used would be appropriate. Both Chapter 2 and Chapter 3 is to be considered as background chapters, describing theoretical studies and the CyberBike's state at the point where the work on this thesis started. The next chapter is discussing some various alternative solutions; which choices were made and what turned out not to work. Then, in Chapter 5, the final solutions are described. A recommendation on further work is put in Chapter 8 *before* the conclusion part in Chapter 9. This feels natural for a project with a considerable part of work left before it has reached its goal.

This thesis is written in English for two main reasons. The first is that the candidate wanted to gain the experience on carrying out a large technical report in English, realizing that the close future as an employee includes a considerable amount of writing on technical issues in this language. And second; the realization of an autonomous bicycle, controlled by using an inverted pendulum substituting a leaning rider, has interests beyond NTNU and Norway.

Chapter 2

Background

2.1 The CyberBike model

As previously mentioned, Bjermeland [2006] has developed a mathematical model of the CyberBike, and a control algorithm that has been tested in simulations on the mathematical model. To be able to connect the work done by Bjermeland and Loftum into a complete functioning system, some understanding of the work done is necessary. This section gives a short summary of the contents of [Bjermeland, 2006].

The task of his thesis was to develop a model, a controller and a simulator for the bicyle, using steering and leaning (the inverted pendulum) as the manipulated variables.

2.1.1 Important assumptions and simplifications

Some assumptions and simplifications were made in the bike-modeling process.

First the bicycle is divided into five rigid bodies:

1. The front wheel
2. The rear wheel
3. The rear frame
4. The front frame (handlebars and front fork)
5. The inverted pendulum

Figure 2.1 shows a simplified model of the CyberBike, where the center of mass for all five rigid bodies are indicated. This assumption neglects elasticity in the frame and other non ideal movements and deformations of the different bodies.

The orientation in space used in the model is shown in Figure 2.2, where the z-axis is pointing upwards, the x-axis is pointing in the bikes running direction when the yaw angle ψ is zero.

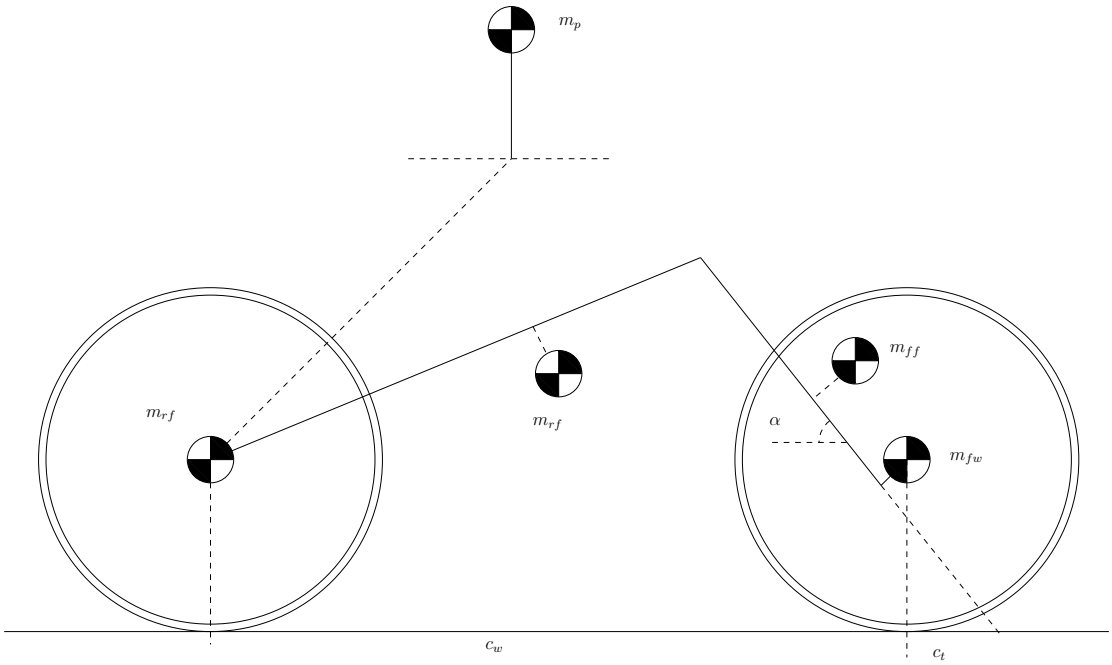

Figure 2.1: Main parameters for the bike model. Figure redrawn from [Bjermeland, 2006].

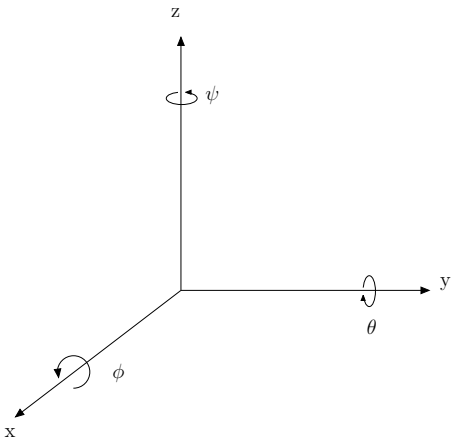

Figure 2.2: Orientation within a right-handed coordinate system.

For small angles, a rotation matrix could be approximated as shown in Equation (2.1):

$$
\begin{aligned}
R_b^a &= R_z(\psi)R_y(\theta)R_x(\phi) \qquad (2.1)\\
&= \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix} \\
&= \begin{bmatrix} \cos(\psi)\cos(\theta) & -\sin(\psi)\cos(\phi)+\cos(\psi)\sin(\theta)\sin(\psi) & \sin(\psi)\sin(\phi)+\cos(\psi)\cos(\phi)\sin(\theta) \\ \sin(\psi)\cos(\theta) & \cos(\psi)\cos(\phi)+\sin(\phi)\sin(\theta)\sin(\psi) & -\cos(\psi)\sin(\phi)+\sin(\theta)\sin(\psi)\cos(\phi) \\ -\sin(\theta) & \cos(\theta)\sin(\phi) & \cos(\theta)\cos(\phi) \end{bmatrix} \\
&\approx \begin{bmatrix} 1 & -\psi & \theta \\ \psi & 1 & -\phi \\ -\theta & \phi & 1 \end{bmatrix}
\end{aligned}
$$

This makes the rotation matrix linear. But it should be kept in mind that the model will give a large error when the system starts to operate at wider angles.

Another important assumption made is that the bike is symmetric at its lengthwise direction while in equilibrium. This is not true for the real CyberBike. As noted by Bjermeland [2006, p. 90]) the placement of the steering motor to the left of the frame, is violating this symmetry. But the introduction of the batteries, which are heavy compared to most elements on the bike, will have a considerable effect on the rear frame's center of mass.

It is also stated that constant velocity is a consequence of the linearized equations.

Another simplification is that the bike is moving on a level surface (i.e. the pitch angle are assumed to be zero at all times).

2.1.2 Modeling the different parts

The position and parameters of the five rigid parts listed in the previous section, are described as rotation and transformation matrices relative to an inertial frame i. Then the rear wheel and rear frame, and the front frame and front wheel are connected together to reduce the number of mass centers from five to three.

The positions of these mass centers are:

Rear frame and rear wheel m_r:

$$
r_{m_r}^i = \begin{bmatrix} x_{rw} + x_r \\ y_{rw} + x_r\psi_r - z_r\phi_r \\ z_r \end{bmatrix} \qquad (2.2)
$$

where:

- x_{rw} and y_{rw} forms the x- and y-component of the vector r^i_{rw}, which is the position of the point where the rear wheel is in contact with the ground surface
- x_r, y_r and z_r are the wheel position of the center of mass.
- ϕ_r is the lean angle of the rear wheel/rear frame from the vertical
- ψ_r is the rear frame's yaw-angle (heading).

Pendulum m_p:

$$r^i_{m_p} = \begin{bmatrix} x_{rw} + x_p \\ y_{rw} + x_p\psi_r - z_p\phi_r - h_p\phi_p \\ z_p \end{bmatrix} \tag{2.3}$$

where:

- x_p and z_p is the position of the pendulum center of mass, relative to the rear wheel to ground contact point
- h_p is the length of the pendulum, from the rotating center, to the pendulum center of mass
- ϕ_p is the pendulum angle

Front frame and front wheel m_f:

$$r^i_{m_p} = \begin{bmatrix} x_{rw} + x_f \\ y_{rw} + u\delta + x_f\psi_r - z_f\phi_r \\ z_f \end{bmatrix} \tag{2.4}$$

where:

- x_f and z_f is the position of the m_f related to the rear wheel to ground contact point
- u is the lenght from the front fork to the m_f center of mass point, measured in a direction perpendicular to the front fork (i.e. $\frac{\pi}{2} - \alpha$ from the horizontal forward (x) axis).
- δ is the steer angle

2.1.3 Other important parameters

To understand how to connect the Simulink model made by Bjermeland to the actual CyberBike, an understanding of some of the various defined variables is needed. Two important vectors used in the model are:

$$q = \begin{bmatrix} q_1 \\ q_2 \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} \phi_r \\ \delta \end{bmatrix} \\ \phi_p \end{bmatrix} \tag{2.5}$$

$$f = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} M_{\phi_r} \\ M_\delta \end{bmatrix} \\ M_{\phi_p} \end{bmatrix} \tag{2.6}$$

where:

- M_{ϕ_r} is the leaning torque on the total system
- M_{ϕ_p} is the leaning torque on pendulum rider
- M_{ϕ_r} is the steering torque

Controller design

The controller is designed to get a desired turn rate of the bicycle. This rate is either measured or estimated through an observer described in [Bjermeland, 2006, chap. 6]. An LQG[1] control strategy was employed, which makes it possible to weigh the different error states of the physical system, and penalize excessive use of the actuators (i.e. motors in the CyberBike's case). A principal block diagram of an LQG controller is shown in Figure 2.3.

The observer states are shown in Equation (2.7) and (2.8), and the state space model of the system is described in Equation (2.10) and 2.11).

[1] Linear quadratic Gaussian (control)

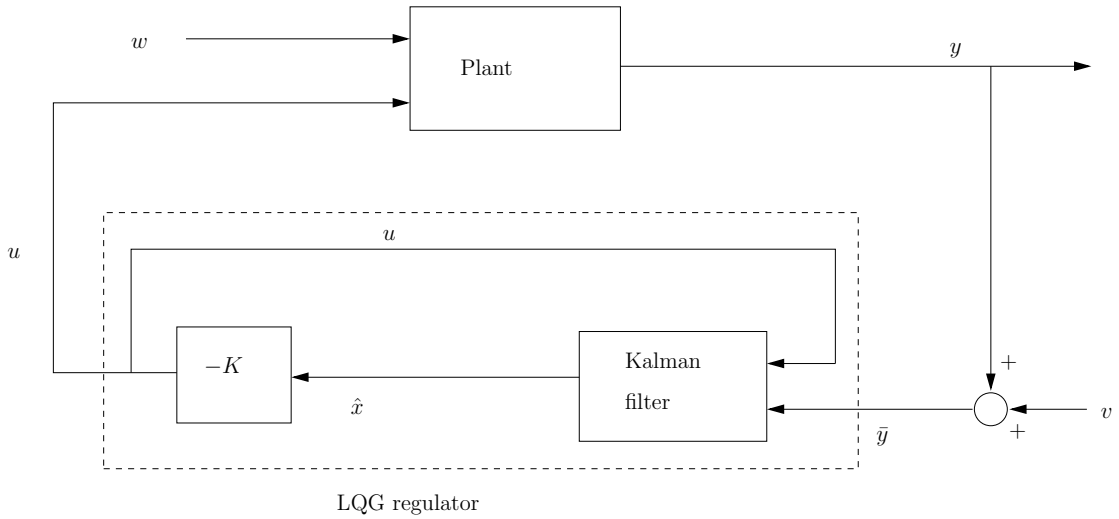

Figure 2.3: LQG control block diagram, redrawn from [Bjermeland, 2006].

$$x_1 = \begin{bmatrix} \phi_r \\ \delta \\ \phi_p \end{bmatrix} \tag{2.7}$$

$$x_2 = \dot{x_1} = \begin{bmatrix} \dot{\phi_r} \\ \dot{\delta} \\ \dot{\phi_p} \end{bmatrix} \tag{2.8}$$

$$u = \begin{bmatrix} M_\delta \\ M_{\phi_p} \end{bmatrix} \tag{2.9}$$

$$\dot{x}_1 = x_2 \tag{2.10}$$

$$\dot{x}_2 = M^{-1}(-(K0 + v^2 K2)x_1 - vC1x_2 + u) \tag{2.11}$$

where

- M is the 3×3 *Mass Matrix*
- $K0$ is the 3×3 *Velocity Independent Stiffness Matrix*
- $K2$ is the 3×3 *Velocity Dependent Stiffness Matrix*
- $C1$ is the 3×3 *Velocity Dependent Damping Matrix*
- v is the speed vector

To trace all parameters in the matrices in Equation (2.11) back to variables introduced in the previous sections, would require more space in this thesis than seems reasonable. Thus; for a further description of the elements in Equation (2.11), it is reffered to [Bjermeland, 2006].

Then the Kalman filter is designed. In [Bjermeland, 2006, p. 60] it is stated that:

> *"It is assumed that the turning rate is estimated through the measurement of the steering angle. The heading state is therefore omitted in the A-matrix, and the turning rate is measured in the C-matrix from the steering angle and angular velocity. This should be altered if the turning rate is measured with other principles, such as a gyro."*

This would have some impact on the solution of this thesis, owing to the fact that a gyro actually *is* used to measure the yaw, see Section 3.1.7 and Section 4.1.3.

Matlab and Simulink model

The main Simulink model made by [Bjermeland, 2006] is shown in Figure 2.4.

To make the model work as it is supposed to, it is necessary to run the files shown in Table 2.1. These files are automatically loaded into the Simulink model at initialization. A simulation GUI[2] is popping up when the simulation is finished (see Figure 2.5). This could be deactivated in the menu that shows up when double clicking on the "Plotting" block (located at the upper right corner in Figure 2.4).

Number in sequence	File	⟶ Generates
1	`load_parameters.m`	⟶ `parameters.mat`
2	`initModel.m`	⟶ `bikesystem.mat`
3	`get_lqg.m`	⟶ `kalman.mat`
		⟶ `lqr.mat`
		⟶ `speed.mat`

Table 2.1: Matlab files to initiate the parameters used in the Simulink model.

The solver used in Simulink for this model is a Runge-Kutta fixed step numerical algorithm [see Egeland and Gravdahl, 2002, chap. 14].

[2] Graphical User Interface

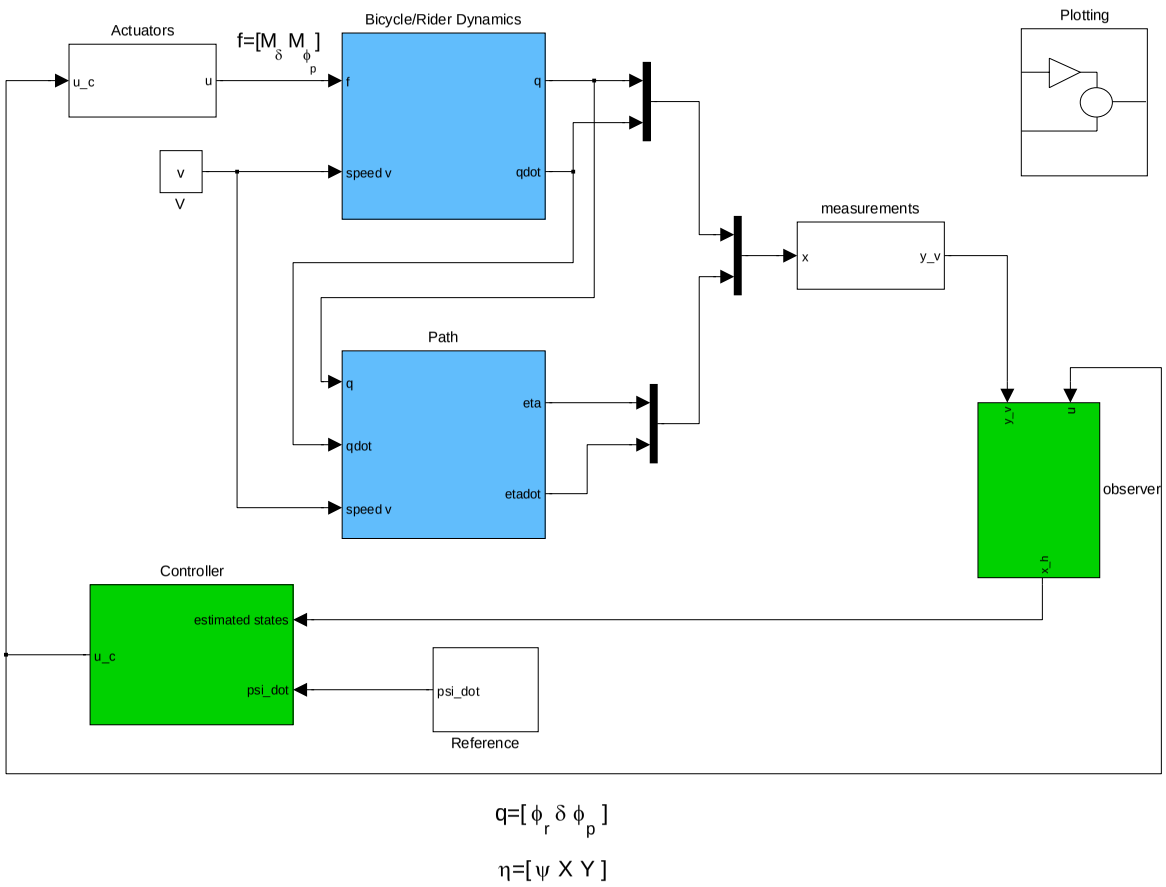

Figure 2.4: Simulink block diagram of the bike model and the control system, made by [Bjermeland, 2006].

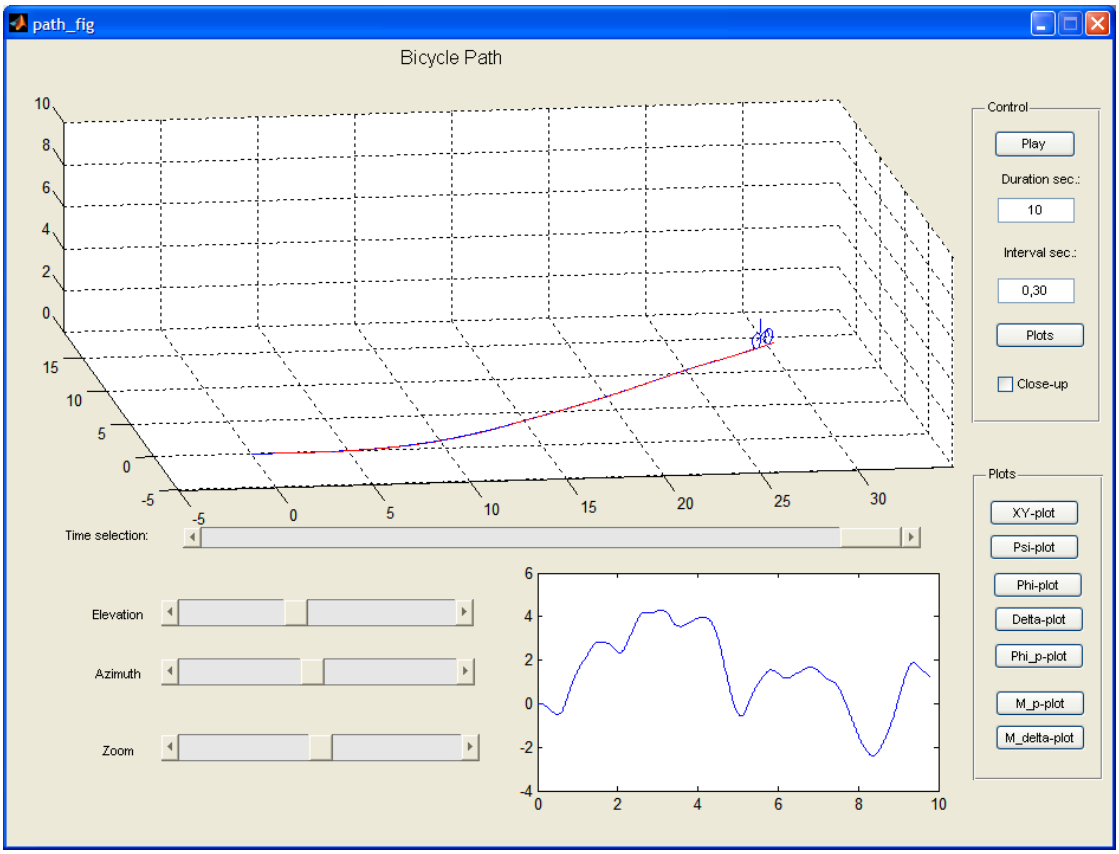

Figure 2.5: Screenshot of the simulation GUI, constructed by Bjermeland [2006].

2.2 QNX Neutrino

QNX Neutrino is an RTOS[3] made by QSSL[4]. The fundamental concepts of QNX Neutrino are *message passing* and *resource managers*, and thus the description in the following two section are mainly in regard to these subjects, based on [Krten, 2001, chap. 2 and 5 respectively].

2.2.1 Message Passing

The QNX family of operating systems has a microkernel architecture, and achieves scalability by making all service-providing components modular. When installing a QNX Neutrino system, its consisting modules should be as independent of each other, that every module could be considered optional. Thus it should be no problem to make the system as small – or big – as possible. The key behind this is the message passing between the modules.

The code, for instance for performing a read or write operation on a file, will look like standard POSIX C. The message passing part is done by the Neutrino C library, which leads to the programmer not having to write message-passing functions if he is developing code for a QNX system.

A great benefit from the message passing architecture is that it is making network distributed systems act all the same as one-node systems. The messages could just as easily be sent to a module on another node as to the same node, as opposed to traditional-kernel systems where local and remote (network) services are implemented in a totally different way.

Modules are divided into servers and clients. The key phrases used by Krten [2001, p 111] are:

- "The client *sends* to the server."
- "The client *receives* from the client."
- "The server *replies* to the client."

When the client wants to send a request to a server, it will get blocked until the server has completed the request and the client has got its "answer". First it has to establish a connection with the *ConnectAttach()* function, which returns a *connection ID (coid)*, and then it could use this ID to send its message by the *MsgSend()* function.

The server has to create a channel, by using the *ChannelCreate()* function. This makes the clients able to connect via *ConnectAttach()*. Then the server will block on a *MsgReceive()*, until it get its request. Then the client will be blocked while the server is processing the request. The answer is sent by *MsgReply()*, which unblocks the client.

[3] Real-Time Operating System
[4] QNX Software Systems Limited

There is a need to follow a strict *send-hierarchy* in a message passing environment. This means that two threads should never send messages to each other. Threads should be organized in levels, where all sends goes from one level to a higher level. This is to prevent deadlocks, because if two threads could send messages to each other, sooner or later they both will get blocked, waiting for the other's to reply.

Sometimes, situations where the send hierarchy would have to be violated arise. Then the non-blocking function *MsgDeliverEvent()* comes into use. As an example, consider a client sending a time consuming request to the server, but is not interested in being blocked while waiting for the server to finish its task. Then it makes a `struct sigevent`, which in turn is used by the server – as an argument in the *MsgDeliverEvent()* call – to inform the client about its completion of the request.

Messages are always delivered in a priority order. That is; if two processes sends a "simultaneous"[5] message, then the entire message from the process with the highest priority is delivered to the server first.

When a message sent by the client is bigger than the buffer specified in the server's receive buffer, the server could call the function *MsgRead()* to read out the rest of the message from the clients address space, while the client is still blocked. Similarly, there exist a function called *MsgWrite()* which makes the server write into the clients address space. This is useful e.g. in cases where the server is a I/O[6] driver, and the client asks the driver for several megabytes of data. Then the server could write it directly into the clients buffer, instead of constantly running with a huge memory allocated for such large requests.

To send multipart messages the *IOV*[7] facilities in QNX could be used together with the corresponding *Msg*v* functions[8] (e.g. *MsgWritev()* and *MsgReadv()*). An IOV contains a number of pointers to memory locations and these *Msg*v()* functions take an IOV and number of the vector parts as arguments, instead of the buffer and its length.

Another mechanism of the message passing facility is *pulses*. This distinguishes itself from the rest by the property that it is *non-blocking*. When a *MsgReceive()* returns 0 as receive ID it means that the message is a pulse. With pulses only 40 bits of content can be sent. In cases where a pulse is the only type of message one want to receive, the *MsgReceivePulse()* comes into play.

Finally it is worth mentioning that QNX Neutrino is doing *priority inheritance* on the message passing threads, to avoid *priority inversion*. These terms might need a short explanation:

Priority inversion is the phenomenon that arises when a low-priority thread is consuming all available (or preventing other threads from consuming) CPU[9] time, even if higher-priority threads are ready to run.

[5] On a single processor machine there is no such thing as absolute simultaneous processing.
[6] Input/Output
[7] I/O vector
[8] The character "*" is here used as a *wildcard*, and could be replaced for instance by "Write", "Send", "Reply" or "Read".
[9] Central Processing Unit

Priority inheritance is the solution provided by Neutrino, and is a feature that makes a receiving thread inherit the priority of the highest of all blocking clients.

For an explanation on how problems of priority inversion could occur, and why inheritance is the solution to the problem, refer to [Krten, 2001].

2.2.2 Resource Managers

Resource Managers (RM) are simply programs with some well-defined characteristics, which have the goal of presenting an abstract view of a service. The abstraction is based on the POSIX[10] specification. RM will almost exclusively be dealing with file-descriptor based functions. Clients that whish to use an RM usually initiates the communication by an *open()* or *fopen()* call. In QNX Neutrino, such a call is routed as messages to the *Process Manager* (PM), which will return which resource manager(s) that suits the *open()* call (i.e. which file was specified as argument). Then a request to the referred RM is sent. When the RM replies, the *open()* call returns.

The Resource Manager itself has to register which part of the pathname space it whish to be responsible for.

The Resource Manager library provided by QSSL consists of some pieces:

- thread pool functions
- dispatch interface
- resource manager functions
- POSIX library helper function

The drivers `devc-imu`, `devc-velo`, `devc-dmm32at` written by Loftum [2006] for the CyberBike, are all made as Resource Managers.

The most important function calls in an RM are:

dispacth_create() creates a dispatch structure which is used for blocking on the message reception.

iofunc_attr_init() initializes the attribute structure used by the device. The structures contain information about a particular device and it exists one per device name.

iofunc_func_init() initializes the two data structures `cfuncs` and `ifuncs`, which contain pointers to the connect and I/O functions, respectively. The library is providing default POSIX versions of functions for handling connect and I/O messages, and the *iofunc_func_init()* is binding these to the given function tables supplied as arguments.

[10] Portable Operating System Interface

resmgr_attach() creates a channel that the RM will use for receiving messages, and talks to the PM to tell it which part of the pathname space this RM is going to be responsible for. This is where the dispatch handle, pathname, and the connect and I/O message handlers all get bound together.

resmgr_context_alloc() allocates a resource managers internal context block, which contains information relevant to the message being processed.

resmgr_block() is the blocking call, making the RM wait for a message from a client.

resmgr_handler() is called once the message arrives from the client, to process the request.

In [Krten, 2001] a figure of the "big picture" is shown, which – according to Krten – *"contains almost everything related to a resource manager"*. A version of this figure is shown in Figure 2.6.

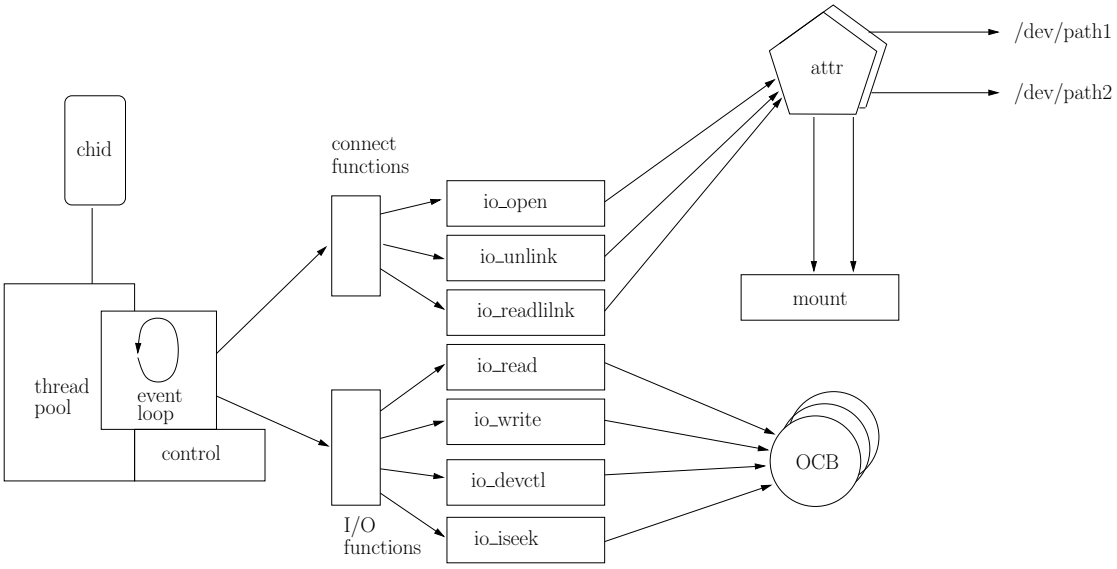

Figure 2.6: Architecture of a resource manager – the big picture. Figure redrawn from [Krten, 2001] (OCB = Open Context Block).

The resource manager receives certain well-defined messages, as can be seen from Figure 2.6, these can be divided into two categories; *connect messages* and *I/O messages*. Connect messages are related to pathname-based operations, and may establish a context for further work. The I/O messages arrive *after* a connect message, and indicate the actual request from the client. QSSL provides a set of POSIX helper functions in the resource manager library, that performs a lot of the work of dealing with these messages. The function *iofunc_func_init()* (mentioned earlier) initializes a list of these to the resource manager, but the functions one might want to give some extra functionality or different behaviour – i.e. the functionality one actually whish to achieve in the RM -

could afterwards be overridden. Here the function calls *iofunc_ *_ default()* (where '*' is meant to be replaced by the corresponding function, e.g. *read, write*, or *devctl*) might come in handy.

The *devctl()* function, mentioned above, is important in regards to the subject of Resource Managers. The function name is an abbreviation for *DEVice ConTroL*, and is used by the client to control aspects of the RM. The *devctl()* function takes the following five arguments:

`fd` is the file descriptor of the resource manager that is receiving the *devctl*-request.

`dcmd` is the command itself, consisting of 2 bits describing the direction of the data transfer (if any), and 30 bits describing the command.

`dev_data_ptr` is a pointer to a data area for sending to, receiving from, or both.

`nbytes` is the size of the `dev_data_ptr` data area.

`dev_info_ptr` is an extra information variable that can be set by the RM.

The CyberBike drivers `devc-dmm32at`, `devc-velo` and `devc-imu` all implements their own version of *devctl()*.

In most of these drivers a set of *dispatch_ context_ alloc(), dispatch_ block()* and *dispatch_ handler()* functions have been used instead of the corresponding *resmgr_ *()* functions, mentioned earlier. For instance the function *resmgr_ block()*, which takes a variable of type `resmgr_context_t` as an argument, is a special case of the function *dispatch_ block()*, which takes a variable of the type `dispatch_context_t` in stead. The *resmgr* versions should only be used with simple resource managers.

Chapter 3

System description

This chapter describes the state of the CyberBike at the point where the work on this thesis started. First the various hardware devices are introduced, and then a short summary of the code that was delivered with the project.

3.1 Hardware

An overview of the CyberBike's hardware units at the starting point is shown in Figure 3.1. Monitors, Ethernet connections, keyboards and mice are not shown. The IMU and propulsion motor are included in the figure, even though they were unmounted and not connected by the start of this thesis, because a considerable effort were put into making them available to the CyberBike before this thesis started.

3.1.1 Power supply

The power supply unit used for DC/DC converting (transformation) is a ACE-890C from IEI Technology Corporation. Specific data for this module is shown in Table 3.1, which is extracted from the vendor's website [IEI Technology Corp., 2005]. Figure 3.2 shows the ACE-890C.

3.1.2 Computer

Figure 3.3 shows the computer used on the CyberBike project, and in Table 3.2 a summary of its specific data, extracted from the vendors website, are listed.

The card is a Wafer-9371A Single-board computer (SBC), with a 400 MHz[1] Intel Celeron processor, 256 MB[2] SO-DIMM[3] RAM[4], and many different possibilities for con-

[1] Mega Hertz
[2] Mega Byte
[3] Small Outline Dual In-line Memory Module
[4] Random Access Memory

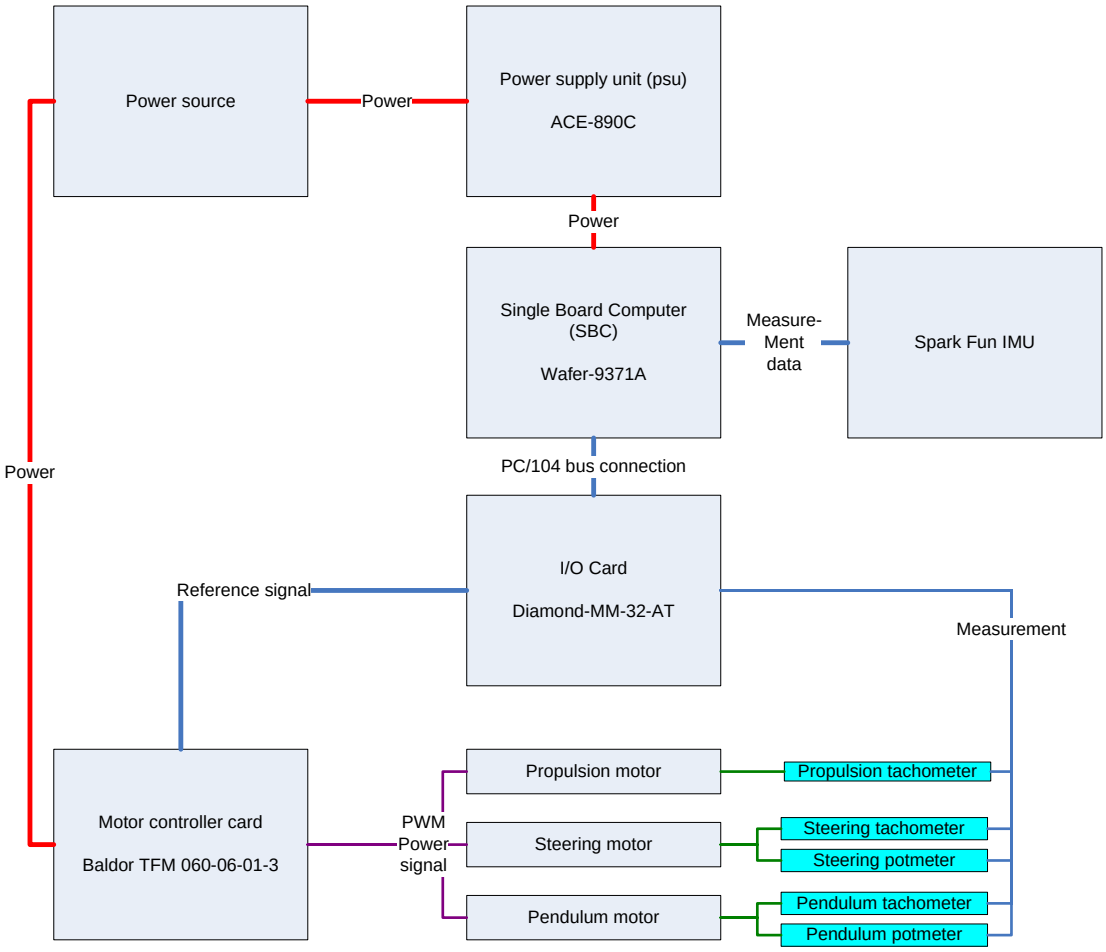

Figure 3.1: Overview of the CyberBike's various hardware devices by the start of this work.

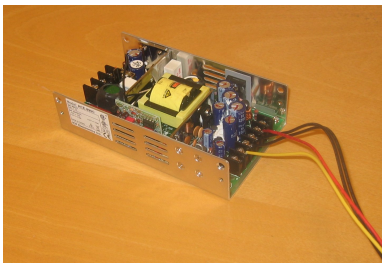

Figure 3.2: The ACE-890C power supply unit. Picture copied from [Loftum, 2006].

Input			
Voltage	18 ∼ 36VDC		
Input Current	7A(RMS)@24VDC		
Output			
Voltage	Min. load	Max. load	Ripple & Noise
+5V	0A	10A	50mV
+12V	0A	2.5A	100mV
-12V	0A	0.5A	100mV
General			
Power	86W		
Efficiency	70 %		
MTBF	251,000hrs		
Temperature	0 ∼ 50°C (Operating)		
	-20 ∼ 85°C (Storage)		
Dimension	$152.4 \times 89 \times 39mm$		

Table 3.1: Specific data for ACE-890C.

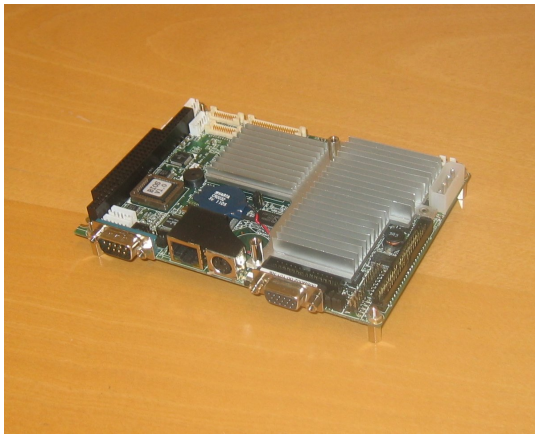

Figure 3.3: The Wafer-9371A SBC used at the CyberBike. Picture taken from [Loftum, 2006].

Parameter	Value
Product	WAFER-9371A
Form Factor	3.5" SBC
CPU	ULV Intel Celeron 400
Display	CRT 36-bit TTL 2x18-bit LVDS
I/O Interface	1x EIDE 1x FDD(optional floppy drive connector) 1x PS2 Keyboard connector 1x PS2 Mouse connector 1x RS-232/422/485 1x RS-232 1x LPT (parallel port connector)
Ethernet	10/100Mbps RTL8100C
USB	2x USB 1.1
Audio	ALC655 5.1CH
IrDA	115kbps
WDT	$1 \sim 255$ sec
Power Consumption	5V@2.01A ULV Celeron 400/256MB
Dimension	5.7" $\times$ 4"

Table 3.2: Specific data for Wafer-9371A.

nection, as shown in the Table 3.2. PC/104 devices, screens, USB[5]-mice, keyboards, USB-sticks, hard drives and serial communication devices can easily be attached to the Wafer-9371A, and communication with the device is made simple by an Ethernet connector.

COM1 is set up to be using RS-232 signals, but COM2 could be configured to communicate on RS-232, RS-422 or RS-485 by modifying the jumper JP4 settings on Wafer-9371A. The location of JP4 is showed in Figure 3.4, and the pin configuration is showed in Figure 3.3.

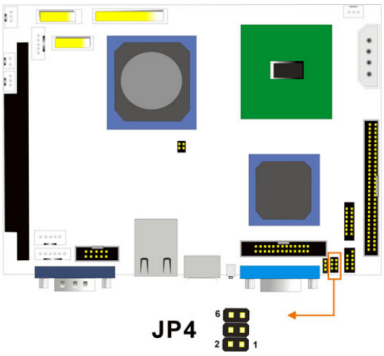

Figure 3.4: The JP4 pin locations on Wafer-9371A. Figure taken from [IEI Technology Corp., 2006].

PIN COMBINATION	DESCRIPTION
1-3	RS-232
3-5/2-4	RS-422
3-5/4-6	RS-485

Table 3.3: The Wafer-9371A JP4 settings to configure COM2.

3.1.3 Storage

When the work on this thesis started, the CyberBike was equipped with a compact flash card (CF) as storage medium, shown in Figure 3.5. This is a 1GB[6] "TwinMOS UltraX CompactFlash" card, with 140x reading transfer speed. It is important that this speed is high, to avoid creating a bottleneck in the execution speed.

[5] Universal Serial Bus
[6] Giga Byte

Figure 3.5: The Compact Flash card from TwinMOS.

3.1.4 I/O card

To be able to read the measurements from the CyberBike's potmeters and tachometers into the CyberBikePC, and to put out a control voltage to the motor controller card, an I/O card is needed. A Diamond-MM-32-AT 16-bit analog I/O module – referred to as DMM-32-AT from now on – connected to the CyberBikePC's motherboard (The Wafer-9371A, see Section 3.1.2) via a PC/104 connector. A picture of the card is shown in Figure 5.1.

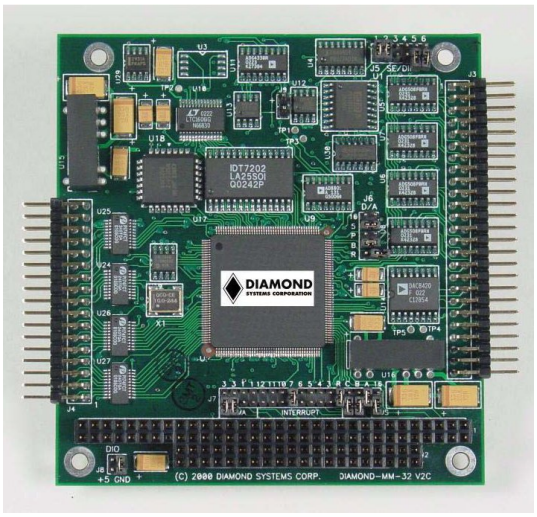

Figure 3.6: The DMM-32-AT-card, taken from [Diamond Systems Corporation, 2003]

The DMM-32-AT's features are listed in [Diamond Systems Corporation, 2003], and the most relevant are listed here:

Analog Inputs

- 32 input channels, may be configured as 32 single-ended, 16 differential, or 16 SE + 8 DI
- 16-bit resolution
- Programmable gain, range, and polarity on inputs
- 200,000 samples per second maximum sampling rate
- 512-sample FIFO[7] for reduced interrupt overhead
- Auto calibration of all input ranges under software control

Analog Outputs

- 4 analog output channels with 12-bit resolution, 5mA max output current
- Multiple fixed full-scale output ranges, including unipolar and bipolar ranges
- Programmable full-scale range capability
- Auto calibration under software control

In addition the DMM-32-AT has 24 bidirectional digital I/O lines, a 32 bit counter/-timer for A/D[8] pacer clock and interrupt timing, and a 16 bit general purpose counter/timer, both with programmable input sources, and multiple-board synchronization capability.

Headers

In this section it is attempted to give a short explanation of the DMM-32-AT's 9 headers, with respect to the most CyberBike-relevant features. An overview of the headers is given in Table 3.4.

The analog I/O pins are located at J3, and the digital ones at the J4 header. The latter is unconnected in the CyberBike's case, but should be ready for use if digital measurement units are to be connected in the future.

The analog input channels are configured as 16 single ended (i.e. referenced to analog ground) plus 8 differential (i.e. measured by subtracting the low input from the high input), by setting the J5 jumpers as shown in Table 3.5. The resulting pinout on the analog out header is shown in Table 3.6. Differential input is useful when common mode noise rejection is desired.

[7] First In - First Out
[8] Analog-to-Digital

Name	Description
J1	PC/104 8-bit bus header
J2	PC/104 16-bit bus header (only used for interrupt level)
J3	Analog I/O header (includes trigger and ctr/timer signals)
J4	Digital I/O header
J5	Analog input single-ended / differential configuration
J6	D/A unipolar / bipolar / full-scale range configuration
J7	Base address / DMA level / interrupt level / bus width
J8	Digital I/O pull-up / pull-down configuration
J9	Test connector; not used in normal operation

Table 3.4: Overview of DMM-32-AT headers. Taken from [Diamond Systems Corporation, 2003, p. 5]

	Jumper settings					
Configuration	1	2	3	4	5	6
0-7 SE, 8-15 DI, 16-23 SE	1	0	0	1	0	1

Table 3.5: Header J5 configuration for analog in on DMM-32-AT

In Table 3.6 the signal *Vref Out* is a +5 volt signal from the on-board reference chip, *A/D Convert* could be used to synchronize multiple boards, *Dout 2 - Dout 0* are digital output ports with counter/timer functions, *Din 3 - Din 0* are digital input ports with counter/timer and external trigger functions, $+5V$ is connected to the PC/104 bus power supply, and *Dgnd* is digital ground connected to the PC/104 bus ground.

Header J7 have jumpers set on position '7' and 'R', which means that a $1K\Omega$ pull-down resistor is connected to the IRQ line 7. Also a jumper is set on 'A', to set the base address to default; 0x300.

Header J6 controls the D/A[9] output, and jumpers are set like shown in Table 3.7 to give a bipolar $\pm 5V$ output range. Other jumper settings could give $\pm 10V$ (bipolar), unipolar $0 - 5V$ or $0 - 10V$, or even programmable unipolar or bipolar [see Diamond Systems Corporation, 2003, p. 10]. The last pair of pins, the "R" position, controls the *"Power-up reset mode"*, and is left open to reset to mid-scale (0V in bipolar mode).

I/O Register Map

The DMM-32-AT occupies 16 bytes in the system I/O address space. In [Diamond Systems Corporation, 2003, chap. 6 and 7] these registers are described. This is interesting when the *"Universal Driver™"* from the vendor is not used, which is the case for the original `devc-dmm32at` driver made by Loftum. Furthermore, some problems on the DMM-32-AT (see Section 6.5) made it necessary to understand how these registers work; hence a summary of these chapters is following next.

[9] Digital-to-Analog

AGND	1	2	AGND
Vin 0	3	4	Vin 16
Vin 1	5	6	Vin 17
Vin 2	7	8	Vin 18
Vin 3	9	10	Vin 19
Vin 4	11	12	Vin 20
Vin 5	13	14	Vin 21
Vin 6	15	16	Vin 22
Vin 7	17	18	Vin 23
Vin 8+	19	20	Vin 8-
Vin 9+	21	22	Vin 9-
Vin 10+	23	24	Vin 10-
Vin 11+	25	26	Vin 11-
Vin 12+	27	28	Vin 12-
Vin 13+	29	30	Vin 13-
Vin 14+	31	32	Vin 14-
Vin 15+	33	34	Vin 15-
Vout 3	35	36	Vout 2
Vout 1	37	38	Vout 0
Vref Out	39	40	Agnd
A/D Convert	41	42	Ctr 2 Out / Dout 2
Dout 1	43	44	Ctr 0 Out / Dout 0
Extclk / Din 3	45	46	Extgate / Din 2
Gate 0 / Din 1	47	48	Clk 0 / Din 0
+5V	49	50	Dgnd

Table 3.6: Header J3 (analog I/O) pinout on DMM-32-AT.

	Jumper settings				
Output Range	10	5	P	B	R
$\pm 5V$	0	1	0	1	0

Table 3.7: Header J6 configuration for D/A output on DMM-32-AT

The I/O register map is shown in Table 3.8. Base address is set to 0x300 in this case, due to the J7 settings mentioned earlier.

Base +	Write Function	Read Function
0	Start A/D conversion	A/D LSB (bits 7 - 0)
1	Auxiliary digital output	A/D MSB (bits 15 - 8)
2	A/D low channel register	A/D low channel register readback
3	A/D high channel register	A/D high channel register readback
4	D/A LSB register Auxiliary	digital input port
5	D/A MSB + channel register	Update all D/A channels
6	FIFO depth register	FIFO depth register
7	FIFO control register	FIFO status register
8	Miscellaneous control register	Status register
9	Operation control register	Operation status register
10	Counter/timer control register	Counter/timer control reg. readback
11	Analog configuration register	Analog configuration reg. readback
12	8254 / 8255 register	8254 / 8255 register
13	8254 / 8255 register	8254 / 8255 register
14	8254 / 8255 register	8254 / 8255 register
15	8254 / 8255 register	8254 / 8255 register

Table 3.8: DMM-32-AT I/O Register Map [from Diamond Systems Corporation, 2003, chap. 6].

The available I/O devices that can be accessed in the last 4 registers in Table 3.8 are depending on the last two bits in the BASE + 8 REGISTER (the MISCELLANEOUS CONTROL/STATUS REGISTER):

00 8254 type counter/timer

01 8255 type digital I/O

10 Reserved

11 Calibration

The control bit settings are shown in front of the device name in the above list. In the CyberBike's case, the last setting is the most interesting, because this is the one used in the initialization procedure of `devc-dmm32at`. The first register (BASE + 12) would then be the data register in an EEPROM[10] read or write operation, or a data register in a TrimDAC write operation. The next (BASE+13) would specify an address in the range 0 to 127 by its 7 LSB[11]. Register BASE + 14 is used to initiate various commands related to auto calibration, and register BASE + 15 is used to set an EEPROM Access

[10] Electrically Erasable Programmable Read-Only Memory
[11] Least Significant Bit

Key to help preventing accidental corruption of the EEPROM contents. BASE + 15 is also used to read back the FPGA[12] Revision Code.

Analog ranges and resolutions

The DMM-32-AT uses a 16-bit ADC[13] for analog input, which provides a resolution of $153\mu V$ per change in LSB when the full-scale range is set to $\pm 5V$, as shown in Equation (3.1).

$$\frac{5V - (-5V)}{2^{16} - 1} = 153\mu V \tag{3.1}$$

The value returned by the A/D converter is a two's complement number in the range -32768 to 32767. Because the input range of the A/D is fixed This is regardless of the input range.

Four analog outputs are provided by a 12-bit DAC[14]. This gives the resolution shown in equation Equation (3.2). The resolution is the smallest possible change in output voltage in this case.

$$\frac{5V - (-5V)}{2^{12} - 1} = 2.44mV \tag{3.2}$$

Auto calibration

The DMM-32-AT is equipped with an octal 8-bit TrimDAC and high-precision, low-drift reference voltages on the board. Whenever the programmer decides to call for it, this circuitry works in conjunction with the driver software to perform an auto calibration. The optimum TrimDAC values for each input range are stored in EEPROM.

For A/D calibration the entire process takes about one second for each input range. The *Universal Driver*™ software provided by the Diamond Systems Corporation, have two specific functions for this purpose; *dscADAutoCal()* and *dscDAAutoCal*, and a standalone DOS program to enable calibration without any programming.

3.1.5 Motor controller card

The motors on the CyberBike are powered through a Baldor TFM 060-06-01-3, see Figure 3.7. A summary of the specific data for this card is extracted from [Baldor ASR, 1988]:

- 4 quadrant operation
- 360 Watts possible continuous output power

[12] Field-programmable gate array
[13] Analog-to-Digital Converter
[14] Digital-to-Analog Converter

- 6A continuous phase current
- 24V nominal DC[15] bus voltage (which makes it possible to operate it from two 12V batteries in series circuit)
- Double eurocard format
- Internal power supply, accepting 24 to 65 V as input
- Differential reference input, to avoid ground loops
- Short-circuit-proof between the outputs, and to ground
- Bandwith from DC (0 Hz) up to 2.5 kHz ($\approx$ 14.7 % of the switching frequency of the pulse width modulated output signal)
- Ca. 80 % efficiency

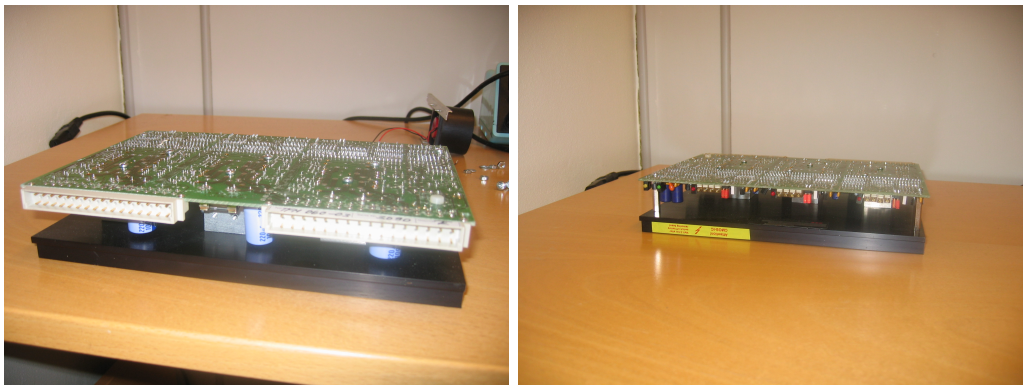

Figure 3.7: The CyberBike's motorcontroller card, a Baldor TFM 060-06-01-3. Seen both from the rear and the front. Pictures copied from [Loftum, 2006].

3.1.6 Motors

There are three motors installed on the CyberBike, to perform three different tasks; make the CyberBike move forward (propulsion), change the steering angle, and change the pendulum angle.

Propulsion motor

The propulsion motor was not installed when the writer of this thesis took over the CyberBike, but it was acquired and available at the start of the project. To be able to connect and use the motor correctly, some information about it was collected, and

[15] Direct Current

a slightly more detailed presentation of this motor than the other two follows in this section.

Figure 3.8 shows the motor from [DtC-Lenze as, 2007], motor type 13.121.55.3.2.0 with worm gear SSN31. The digits in the motor type code have a special meaning [Lenze, 2002]

13.: small DC motor

12 : permanent magnet motor with smooth housing

1.: specification for A side: for worm gearboxes (SSN)

55.: describes the motor frame size

3.: gearbox size 31 / SSN31

2.: motor frame form: flange mounting

0.: Specification B-side: no attachments (e.g. brakes, tachometers or pulse encoders)

Technical specification is shown in Table 3.9.

	Name	Variable	Value	Unit
Motor	Rated power	P_r	200	W
	Rated torque	M_r	0.64	Nm
	Moment of inertia	J	3.2	$kg\ cm^2$
	Rated rotational speed	n_r	3000	rpm
	Outer diameter	d_out	80	mm
	Motor weight (mass)	m_{mot}	3.7	kg
	Rated current	I_r	11.8	A
	Armature resistance	R_A	0.19	Ω
	Permissible radial load	F_R	340	N
	Permissible peak current	I_{max}	77	A
Gear	Max continuous torque	M_{max}	16	Nm
	Rated output torque	M_2	2.7	Nm
	Ratio	i	5	
	Operating factor	c	5.15	

Table 3.9: Specification for the propulsion motor and gear.

Motor for the inverted pendulum

The pendulum motor is a ITT GR 63 x 55 TG11, according to [Loftum, 2006]. Some specific data is given in Table 3.10. On the motor it is mounted a planetary gear with ratio 79:1.

Figure 3.8: The propulsion motor; a Lenze Worm Geared motor

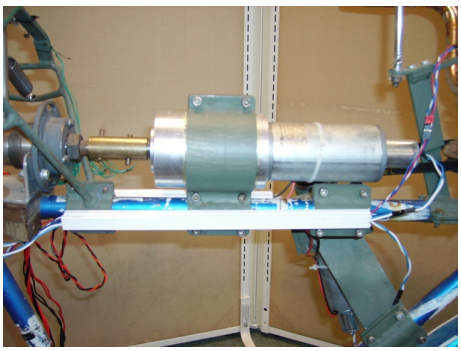

Figure 3.9: The pendulum motor; an ITT GR 63 x 55 TG11 with planetary gear. Picture taken from [Fossum, 2006].

Supply voltage	$\pm 24V$
Nominal current	$4A$
Nominal rotating speed	$3350min^{-1}$

Table 3.10: Specific data for the pendulum motor, read from its nameplate.

Steering motor

The steering motor is an ITT GR 53 x 58 TG11. Its specific data is given in Table 3.11.

Figure 3.10: The steering motor; an ITT GR 53 x 58 TG11. Picture taken from [Loftum, 2006].

Supply voltage	$24V$
Nominal current	$2.9A$
Nominal rotating speed	$3000min^{-1}$

Table 3.11: Specific data for the steering motor, read from its nameplate.

3.1.7 IMU - Spark Fun

The IMU[16] (see Figure 3.11) consists of a small 2x2 inch motherboard, and 3 gyro cards (see Figure 3.12). The motherboard contains a PIC16F88 μC[17] with a built in DAC, and a CD74HC4067 multiplexer. The multiplexer is collecting the 5 measurements provided by each of the 3 gyro cards, and sends them to the μC, which in turn sends them out on its UART[18].

When the IMU receives an ascii character '7' it starts to deliver data over the serial connection. The 34 bytes messages are delivered in a speed of $23.5Hz$, and have a

[16] Inertial Measurement Unit
[17] microcontroller
[18] Universal Asynchronous Receiver Transmitter

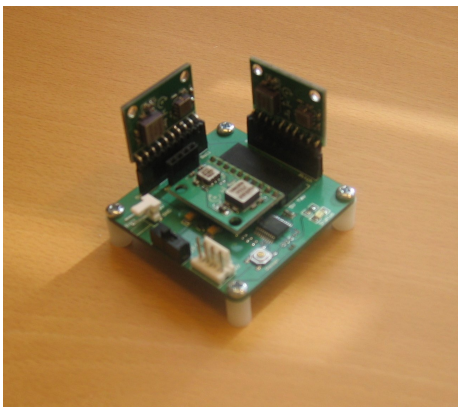

Figure 3.11: IMU with 6 Degrees of Freedom from Spark Fun Electronics. Picture is copied from [Loftum, 2006]

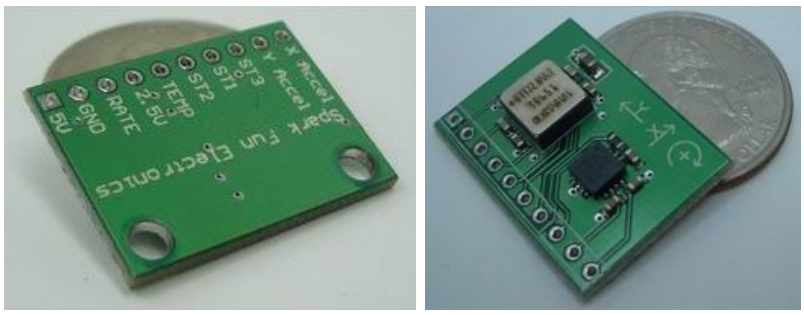

Figure 3.12: Gyro card for the IMU shown from both sides, from [Spark Fun, 2005].

specified sequence. The data stream begins with the ascii character 'A' followed by the 16 DAC measurements (5 measurements from 3 units, plus battery voltage), shown in the following list:

1. Pitch, Rate out
2. Pitch, 2.5 V
3. Pitch, Temperature
4. Pitch, YFilter
5. Pitch, XFilter
6. Roll, Rate out
7. Roll, 2.5 V
8. Roll, Temperature
9. Roll, YFilter
10. Roll, XFilter
11. Yaw, Rate out
12. Yaw, 2.5 V
13. Yaw, Temperature
14. Yaw, YFilter
15. Yaw, XFilter
16. Battery Voltage

The data stream is ended with an ascii 'Z', providing a simple way to synchronize the data stream.

3.2 Software

The operating system running on the CyberBikePC at the starting point of this thesis' work, was a minimal QNX Neutrino OS[19] image, set up by [Loftum, 2006]. Only the most important features of the operating system was included, in order to keep the system simple and small. For instance; no graphical user interface – usually *Photon* in the QNX case – was installed.

Some drivers were made to provide measurement data to the control algorithm. These are shown in Figure 3.13. The drivers are made as resource managers, each providing a set of device file names under the `/dev/` directory, see Section 2.2.2.

[19] Operating System

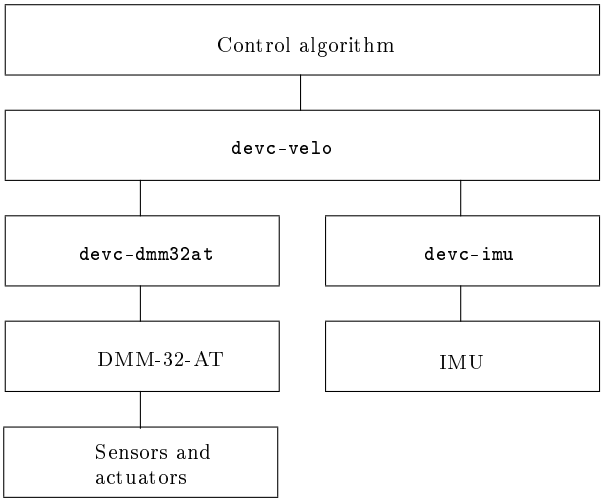

Figure 3.13: Schematic overview of drivers developed by Loftum [2006]. Figure translated and redrawn with modifications from his master thesis.

Chapter 4

Suggested solution

4.1 IMU alternatives

Information about how the CyberBike is moving is intended to be delivered to the control system by an *Inertial Measurement Unit* (IMU). An IMU from Spark Fun [2005] have already been bought by the Department of Engineering Cybernetics for this purpose. By the start of this work the IMU had just arrived from Spark Fun, due to some reconditioning work.

Hence the first thing that had to be done was to test the unit. An IMU driver had already been developed by Loftum [2006], but not thoroughly tested. The IMU has a UART-interface, and this signal had to be transformed to a RS-232-signal to be able to communicate with the CyberBikePC.

4.1.1 RS-232 to UART transceiver

The firmware on the IMU is using the UART at baud rate of 57600bps.

The UART signals delivered from the IMU needed to get transformed to an RS232 signal, to be connected to the Wafer-9371A. For this purpose Maxim/Dallas Semiconductors [2006] is providing the iC *MAX233CPP*, which is placed on its own PCB[1].

Over this connection the output data of 34 bytes is delivered to the CyberBike's computer. A picture of this PCB is shown in Figure 4.1, and the layout could be found in Appendix F.

4.1.2 Driver issues

On the driver for the initial IMU from Spark Fun [2005], some problems were discovered when the driver was started on the CyberBikePC in verbose mode, like shown in Printout 4.1.

[1] Printed Circuit Board

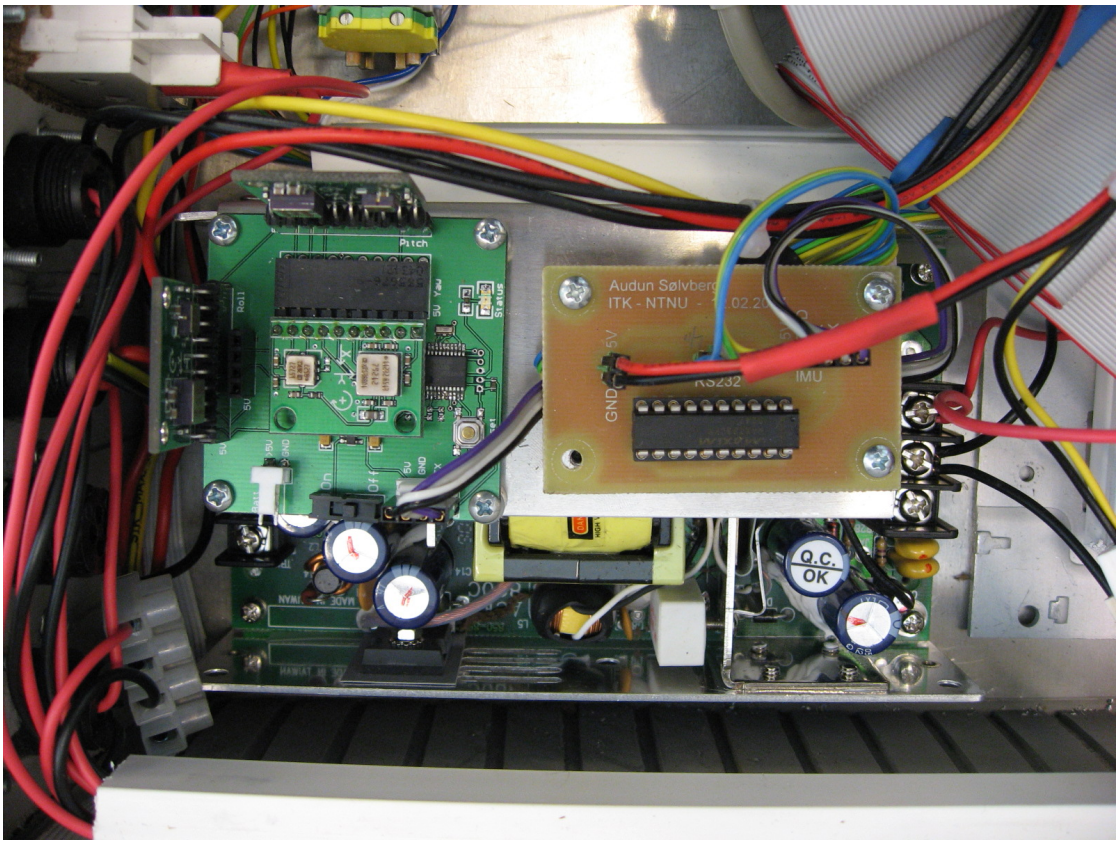

Figure 4.1: Photo of the IMU and *UART-to-RS232-converter* card, mounted in the CyberBike's suitcase.

```
audun@audislappis:~/report$ telnet 129.241.154.76
Trying 129.241.154.76...
Connected to 129.241.154.76.
Escape character is '^]'.
login: root
# devc-imu verbose &
[1] 57360
# devc-imu:     Verbose mode
devc-imu:       Opening serial port ...
devc-imu:       Port opened. File desc: 3
ImuThread:      I'm alive! Filedesc: 3
ImuThread:      Polling...
devc-imu:       Running...
ImuThread:      Read: 522 522 522 522 522 522 522 522 522 522 522 522 522 522 522 522
ImuThread:      Polling...
```

Printout 4.1: Terminal dump of CyberBike-login and starting of IMU-driver.

From there the terminal displays measurements only occasionally, and not in the speed at $23.5Hz$, as stated by Spark Fun [2005]. Nevertheless, the diode on the IMU is still lit, indicating that the unit is in operation.

The first thing to check is if the IMU has ceased delivering the specified amount of measurements. The IMU was connected to the Windows XP development host PC's serial port (RS232) to see if something was wrong with the unit. In Figure 4.2 the output of the terminal window is shown.

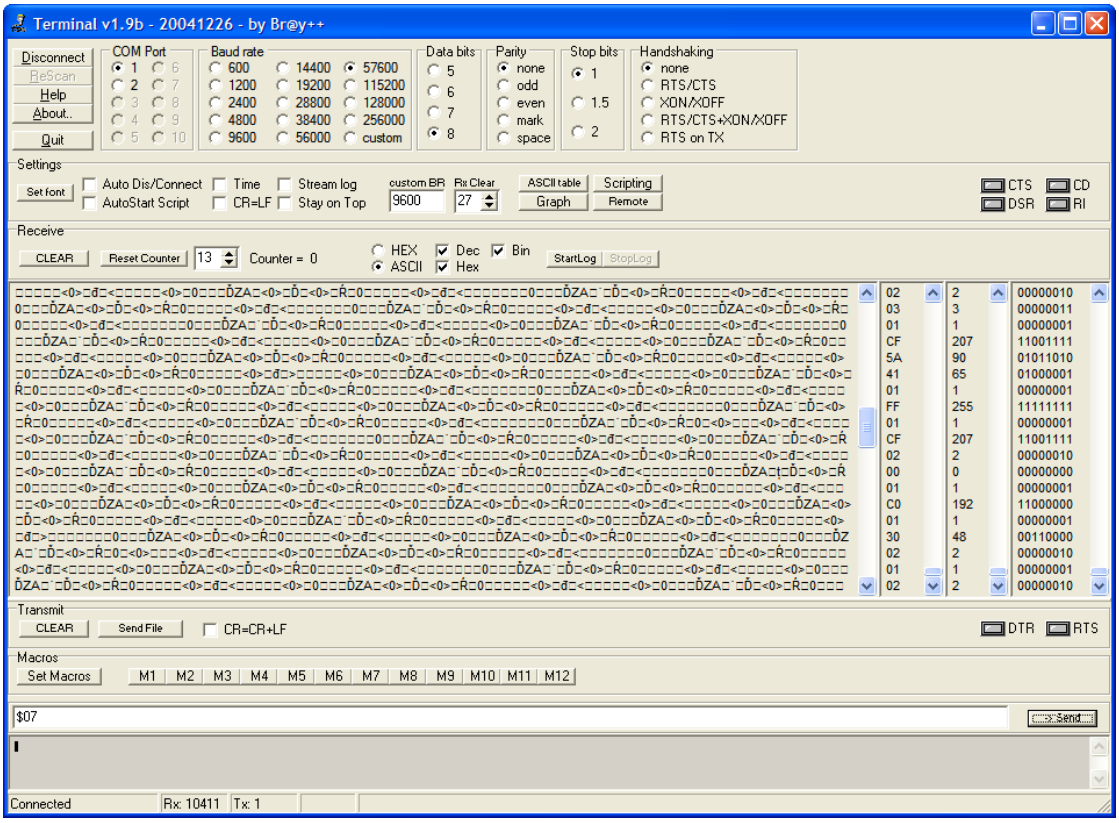

Figure 4.2: Screenshot of terminal window, listening to the IMU.

The terminal has problems coping with the speed of the IMU, and the output on the screen is lagging somewhat. When the IMU is reset by its own hardware button, it takes some time before the terminal window has flushed its buffer to the screen. This is assumed to be an indication that the IMU is delivering data at the specified speed. But here it is important to note that the IMU is supposed to send an ascii character 'A' before its 16 measurements and the character 'Z', see Section 3.1.7. From the terminal window it is difficult to see whether the IMU is delivering the correct data stream – because the terminal interprets the measurements as ascii characters on the screen – but we clearly can see occurrences of 'ZA', between all the "nonsense". This indicates the end of a stream, and the beginning of the next.

Later, when some adjustments of the IMU driver, `devc-imu` was tested, the serial driver at the CyberBike `devc-ser8250` had to be restarted (for a different reason). Afterwards, when the IMU driver was restarted, it began to flush loads of measurements to the telnet terminal (similar to the last two lines in Printout 4.1). Hence something is wrong about the way the serial driver is started at system boot.

In [Loftum, 2006, p 26] the buildfile for the OS image originally running at the CyberBike is shown. Here we can see that the serial driver is executed with the option "-e", which means that the stream is ASCII formatted. The default option "-E" means raw output, and this explains why the IMU driver "suddenly worked" when the serial driver was restarted without specifying any options.

Code sample 4.1 Part of the file buildfile `velo.build` made by Loftum [2006] where specification of how the serial driver `devc-ser8250` should be started during system boot.

```
###############
# Script
###############

[+script] .script={
display_msg "QNX on Cyberbike"
seedres

devc-ser8250 -e -b57600 &
reopen /dev/ser1
```

This problem was part of the motivation for doing changes to the OS running at the CyberBikePC. When the OS upgrade was done – as described in Section 4.2 – extended functionality could be used to investigate the IMU driver further. The program `qconn`, which was started at the CyberBikePC, made it possible to monitor the CPU consumption for each process there, through the Momentics IDE[2] running on the Development PC (Windows XP).

When the serial driver was running alone it practically didn't use any CPU, but as soon as the IMU driver was started, they used – together with the driver for terminal I/O flushing printouts to the screen – 100% of the available processing power. This is a problem because the `devc-velo`, `devc-dmm32at` and the control algorithm (RT-Workshop generated code) needs a significant part of the CPU time, when the bike is operating.

Some code optimizations were performed, with practically no seemingly improvements. The error was finally found to be a wrong argument passed to the IMU driver, when opening the serial device.

The code for opening the serial device in `devc-imu.c` is shown in Code sample 4.2.

[2]Integrated Development Environment

The flag O_NDELAY is specified as a parameter in the *open()* call, making the message passing operations (*read()* and *devctl()*)to this device *non-blocking*. Hence the reading thread in the IMU driver, would check if there was any available data at the serial port, and if not; check again immediately. Practically this thread was performing a *busy-loop*, doing less useful operations most of the time. This also explains why the code optimization didn't give any results; the busy loop would eat up all the freed CPU time.

Code sample 4.2 Opening of the serial device in `devc-imu`. O_NDELAY is specified.

```
printf("%s:\tOpening serial port ...\n",NAME);
if ( (serialFD=open(SERIAL_DEVICE,O_RDWR | O_NOCTTY | O_NDELAY)) == -1 ){
  printf("%s:\tUnable to open port \n",NAME);
  exit(1);
}
else{
  printf("%s:\tPort opened. File desc: %d\n",NAME,serialFD);
  tcsetattr(serialFD,TCSAFLUSH,0);
}
```

Changing the second code line in Code sample 4.2 as shown in Code sample 4.3, solved the problem.

Code sample 4.3 Opening of the serial device in `devc-imu` correctly.

```
if ( (serialFD=open(SERIAL_DEVICE,O_RDWR | O_NOCTTY )) == -1 ){
  ...
```

The CPU consumption for the serial driver and the IMU driver in common, was now reduced from 100% to approximately 14% as shown in the "old version" column in Table 4.1. The "new version" percentages is when running the imu-driver whith the improvements, mentioned above.

CPU consumption	old version	new version
`devc-imu`	2.5%	0.05%
`devc-ser8250`	11%	1%

Table 4.1: CPU consumption for the old and new version of `devc-imu` (approximate values).

These improvements were based on the use of the function *readcond()* instead of *read()*.

The IMU driver is made by two threads running in loops; the first one is handling the requests required by a QNX resource manager, and the second one is reading from the serial device, and stores the values in a measurement array. The first loop is shown in Code sample 4.4.

Code sample 4.4 The first loop in `devc-imu`.

```
while(running) {
  if((ctp = dispatch_block(ctp)) == NULL) {
    fprintf(stderr, "%s:\tblock error\n", NAME);
    return EXIT_FAILURE;
  }
  dispatch_handler(ctp);
}
```

This shows some of the elegance of the QNX resource manager utility. The *dispatch_ block()* function call is blocking until someone performs a call to the IMU driver, such as *read()*, *write()* or *devctl()*. Then the default function for the requested operation is called, unless a new function is registered for that operation. In this case only a new *read()* function (*imu_ read()*) and a *devctl()* (*imu_ devctl()*) function is registered. For instance, if someone is trying to read the value from the roll gyro unit, like shown in Printout 4.2, the *dispatch_ block()* call will unblock, and the function *imu_ read()* will be called. This function is returning the requested value from the measurement-array to the terminal. Access to the measurement array is synchronized by a POSIX mutex `val_mutex`.

```
# cd /dev/imu/roll
# cat rateout
512
512
512
516
519
540

(... continues until hitting "ctrl+c" ...)
#
```

Printout 4.2: Reading the roll rateout from the IMU device.

The second loop is reading from the serial device. The old version (i.e. the code as it was by the starting point of the work on this master's thesis) is shown in Code sample 4.6.

The first thing to note about this code is the *"busy while loop"* at the start, looking for an 'A' in the measurements read from serial device (the variable `fd` is a file descriptor passed to the thread in the *pthread_ create()* call, and is the same as the `serialFD` variable in Code sample 4.3).

The second thing is the synchronization mechanism, a POSIX mutex, which is locked at the start of the loop, and unlocked afterwards. This lock's purpose is to protect

Code sample 4.5 The second loop in `devc-imu`, old version.

```
while (running){
  // Waiting for start character (A)
  if (verbose){
    printf("ImuThread:\tPolling...\n");
  }
  while ( ((iReadCnt = read(fd, &buffer, 1)) <= 0) || buffer!='A');
  // Lock mutex
  if (pthread_mutex_lock(&val_mutex)){
    printf("ImuThread:\tCouldn,t lock mutex.\n");
  }
  // For each of the 16 16-bits measurements:
  // Fetching 2 8-bit values for each measurement,
  // and shifting/putting them into the measurement array.
  if (verbose){
    printf("ImuThread:\tRead: ");
  }
  for (ii=0; ii<16; ii++){
    while ( (iReadCnt = read(fd, &buffer, 1)) <= 0);
    measurements[ii] = ((buffer << 8) & 0xFF00);
    while ( (iReadCnt = read(fd, &buffer, 1)) <= 0);
    measurements[ii] |= (buffer & 0x00FF);
    if (verbose){
      printf("%d ",measurements[ii]);
    }
  }
  if (verbose){
    printf("\n");
  }
  // Release mutex
  if ( pthread_mutex_unlock(&val_mutex)){
    printf("ImuThread:\tCouldn,t unlock mutex!\n");
  }
}
```

the `measurement[]` array from being read from (by the *imu_read()* function mentioned above) at the same time as it is written to by this thread.

When the 'A' is found, the loading of data into the measurement array is done by (at least) 32 sequential *read* operations on the serial device.

The reason for using the *readcond()* function instead of *read()* was because an recommendation from the QNX online library reference [QSSL, 2007c] about *readcond()*:

> *"This function is an alternative to the read() function for terminal devices, providing additional arguments for timed read operations. These additional arguments can be used to minimize overhead when dealing with terminal devices."*

4.1.3 IMU - Xsens (MTi)

At ITK[3] a more advanced (and more expensive) IMU was acquired mainly to be shared among two master projects; the CyberBike and a project on development of an AUAV[4] [see Bjørntvedt, 2007].

The "MTi" IMU from Xsens Motion Technologies is shown in Figure 4.3.

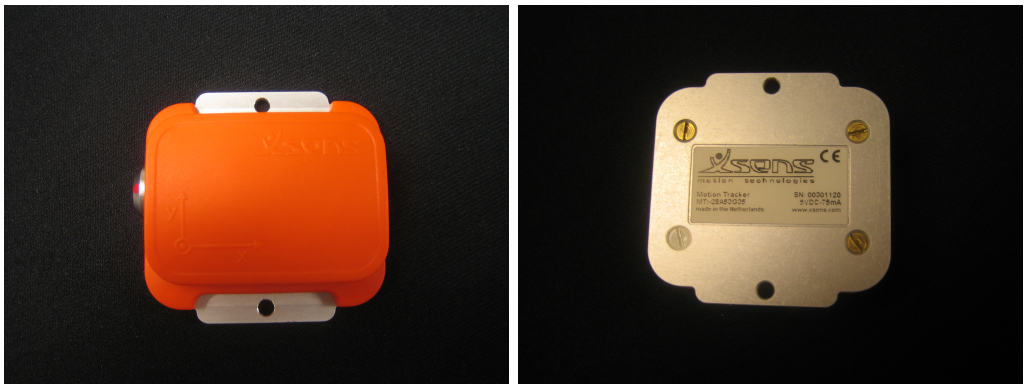

Figure 4.3: The "MTi" from Xsens, with integrated DSP and magnetometer

In [Xsens, 2006, chap. 1.1] it is stated:

> *"The MTi is a miniature, gyro-enhanced Attitude and Heading Reference System (AHRS). Its internal low-power signal processor provides drift-free 3D orientation as well as calibrated 3D acceleration, 3D rate of turn (rate gyro) and 3D earth-magnetic field data. The MTi is an excellent measurement unit for stabilization and control of cameras, robots, vehicles and other equipment."*

[3] Department of Engineering Cybernetics
[4] Autonomus Unmanned Aerial Vehicle

Code sample 4.6 The second loop in `devc-imu`, new version. Most of the error checking and terminal printouts are removed in this sample to increase readability.

```
// Setting up the FORWARD qualifier for the serial device:
if (tcgetattr( fd, &termio ) == -1 ) {
  perror("ImuThread:\tERROR when tcgetattr():\n");
}
termio.c_cc[VFWD] = 'Z';
if (tcsetattr( fd, TCSANOW, &termio ) == -1 ) {
  perror("ImuThread:\tERROR when tcsetattr():\n");
}

while (running) {
  if ( warningCnt > 500 ) {
    printf("ImuThread:\tWarning: high warning limit.\n");
    warningCnt = 0;
  }
  // tell the serial driver (fd) to return data in buffer
  // (within specified size) and that it should be:
  //       - at least 34 characters, or
  //       - 1/10 second before timeout (a 'Z' is dropped/missing)
  iReadCnt = readcond( fd, buffer, sizeof(buffer), 34, 0, 0 );

  // A lot of error checking on iReadCnt not shown here.
  // Errors handled by printing an error message, incrementing a "warning counter" variable
  // and execute a "continue;" to start at the top of the while loop again.

  // Find the 'A' first:
  bi = 0;
  while ( buffer[bi] != ,A, ) {
    bi++;
  }
  if ( (bi > iReadCnt - 34) || (buffer[bi+34-1] != 'Z') ) {
    // something is missing in the frame. Do error handling.
  }

  // Lock mutex
  if (pthread_mutex_lock(&val_mutex)) {
    printf("ImuThread:\tCouldn,t lock mutex.\n");
  }
  bi++; // bi should now point to the element after 'A'

  // When measurement is int:
  for ( ii = 0 ; ii < 16; ii++ ){
    measurements[ii] = ((buffer[bi]<< 8) & 0xFF00);
    measurements[ii] |= (buffer[bi+1] & 0x00FF);
    bi += 2;
  }
  // Release mutex
  if ( pthread_mutex_unlock(&val_mutex)) {
    printf("ImuThread:\tCouldn,t unlock mutex!\n");
  }
}
```

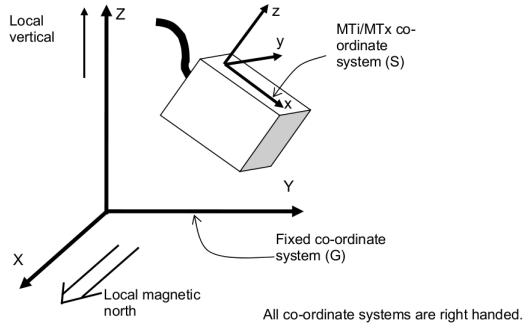

Figure 4.4: The MTi's S-system shown relative to the G-system. Figure copied from [Xsens, 2006].

The MTi is delivered with cables for RS-232 and USB communication (the latter have a serial to USB converter attached to the cable), some example applications, and a "MT Software Development Kit". This includes a C++ class for low-level communication, which is the most important feature for this project. Direct low-level communication with the MTi via RS-232/422 is recommended when real-time requirements are presented. An example code for this kind of communication, which is compatible with and compileable in both for Windows and Linux, is provided with the MTi. This makes development of the driver for this device easier (see Section 5.2.1).

Output from the MTi is given from a calculation of the orientation between a sensor-fixed (S) and an earth-fixed (G) reference co-ordinate system. The local earth-fixed reference system is defined as a right handed Cartesian co-ordinate system with:

- X positive when pointing to the local magnetic North
- Y positive pointing westward; according to the previous point
- Z positive along the vertical axis

The output reference co-ordinate system could be reset to an object-fixed reference system in four different ways:

1. Heading reset; redefines the global coordinate's x-axis, while the z-axis is maintained along the vertical.
2. Global reset; resets all global axes to point in the direction of the current S-coordinate system.
3. Object reset; defines how the sensor is oriented with respect to the coordinate axes to which it is attached.
4. Alignment; a combined object/heading reset.

The data could be received as rotation matrices, Euler angles (roll-pitch-yaw) or quaternions. It is important to note that choosing Euler angles could cause some trouble due to a mathematical singularity when the pitch angle is close to $\pm 90°$, but in the CyberBike's case this should not be an issue.

Connection

As mentioned earlier, the MTi could be connected using both USB and RS-232. Due to some problems with the serial ports (described in Section 6.3), attempts to make the MTi communicate with the CyberBikePC via USB seemed like a good option. A noticeable amount of effort was put into this task, but with no positive result.

Without going too deep into the details of this part, it could be summarized as follows. First, a search for virtual com port drivers in the QNX system and on the internet was done, without finding anything of particular interest. But some references to the QNX USB Driver Development Kits (DDKs from now on) were found.

With a good framework as a basis, the development of a custom made driver for the USB-to-serial converter seemed like an manageable task. The DDK for a USB printer was downloaded from the QNX web page [QSSL]. The web page provides three different USB DDKs; mouse, keyboard and printer. Bulk transfer mode endpoints are only used by the printer DDK, and therefore this was the most attractive starting point.

The information about the device, configuration, classes, subclasses, interfaces, and endpoints for the USB device from Xsens was obtained using the command shown in Printout 4.3.

```
# usb -vvv | less
```

Printout 4.3: Listing the connected USB devices on a QNX system.

This command shows that the device has three endpoints; one control, and two for bulk transfer (one for in and one for out transmissions).

The DDK was imported to the QNX Momentics IDE, and copied to a new project to be configured for the specific purpose. The code was altered to only wait for incoming connections from the Xsens' registered vendor id for, its specified product and vendor specific protocol. So far everything was working; when the device was plugged into the USB port on the CyberBikePC, the *insertion()* function was called, and tried to set up the device. But from there things got complicated. Receiving any data from the device was never accomplished in any way. Seeking help on [OpenQNX], by reading a large part of the USB-driver related threads, gave no result. However; a lot of people seemed to be dealing with the same problem, and by this it was realized that the task was too big to be solved within a reasonable amount of time in this project.

In the final solution the serial connection option was used instead of the USB cable.

4.2 OS upgrade on the CyberBikePC

By the beginning of the work on this master's thesis, the CyberBikePC were running a QNX Neutrino OS image, as described in Section 3.2. This image, developed by Loftum [2006], was minimal, and just enough to get the CyberBikePC online and to run its programs. Some enhancements to the image was desired, to make development and testing a bit easier. The desired features to get into an upgraded system were:

qconn: the utility which makes connection from an IDE on a remote PC possible (i.e. the Windows XP PC used for developing in this project). By using `qconn`, drivers for the CyberBike could be tested and debugged on the bikes own CPU while developed on another computer.

pidin: lists threads running on the node, and which state they are in.

devc-ser8250 options: the serial driver was executed with wrong options specified in the boot-script (see Section 4.1.2), and this should be corrected in this version of the OS image.

updated drivers: some adjustments on the drivers have been performed, and the updated versions should be put on the new image.

devc-gps: driver for the GPS[5].

wireless-ethernet-driver: driver for the wireless Ethernet card.

keyboard setup: the keyboard connected to the CyberBike has a Norwegian layout, but the CyberBikePC is assuming it is English, which makes typing a bit difficult (i.e. the letters on the actual keyboard does not correspond with the letters appearing on the screen while typing).

.kshrc: the developer of the new OS image is familiar with Linux and Bash[6] terminal setup. Hence some settings to be performed during boot were preferable, to make the ksh[7] (shipped whith QNX Neutrino) to appear a bit more like Bash, e.g. TAB-completion and some aliases (ls='ls -F' and ll='ls -l').

other additional commands: common Linux/Unix and QNX programs/commands to make life easier:

- grep (print lines matching a pattern)
- find (search for files in a directory hierarchy)
- less (similar to more, but enables backwards movements in the file)
- ln (make links between files)
- waitfor (wait for a name to exist (QNX))

[5] Global Positioning System
[6] Bourne-Again SHell
[7] Korn SHell

4.2.1 First attempt: New OS image

The buildfile `velo_v2.build` (see appended CDROM, Appendix A) was made to include the changes mentioned in the list above. Then the commands given in [Loftum, 2006, Section 4.2 and 5.2.1] were executed like shown in Printout 4.4. This generates loads of output, not shown here. No error messages were found in the output.

```
# mkifs -v velo_v2.build velo_v2.ifs
```

Printout 4.4: Making OS image from buildfile.

Then the Compact Flash (CF) was removed from CyberBikePC and put into a USB card reader. Having some hard times finding the card on the QNX host workstation, the card reader was plugged into a laptop (running Ubuntu Linux) to be investigated. Printout 4.5 shows the terminal input/output from this, indicating that the card is working as it should.

```
audun@audislappis:/dev$ sudo mkdir /mnt/flash
audun@audislappis:/dev$ sudo mount -t qnx4 /dev/sdb /mnt/flash
audun@audislappis:/dev$ ls -al /mnt/flash
total 1928
drwxrwxr-x 3 root root    4096 2006-06-05 08:59 .
drwxr-xr-x 9 root root    4096 2007-03-14 16:09 ..
-rw------- 1 root root       0 2006-06-05 08:59 .altboot
-r--r--r-- 1 root root  251904 2006-06-05 08:59 .bitmap
-rw------- 1 root root 1705616 2006-06-05 08:59 .boot
-r--r--r-- 1 root root    8192 2006-06-05 08:59 .inodes
-r--r--r-- 1 root root       0 2006-06-05 08:59 .longfilenames
```

Printout 4.5: Mounting CF card on Linux.

Plugged back into the QNX host machine, the card reader with the CF card inserted had to be initialized. This was done by the commands shown in Printout 4.6.

An important note here is that `io-usb` was started with the *"-duhci"* option, as opposed to *"-dehci"*, suggested by Loftum [2006]. The earlier mentioned problems on finding the CF card on the QNX machine arose because of this difference. The reason is probably that another card reader was used in this case, than in Loftum's.

When the CF card was inserted into the CyberBikePC again, a message like shown in Printout 4.8 appeared on the screen. This message is similar to the one described in [Loftum, 2006, p 59], and means that no OS-signature is found.

Loftum solved this by running the command shown in Printout 4.9, but that approach did not solve the problem this time.

```
# io-usb -v -duhci &
[1] 1339429
# devb-umass
# Path=0 - QNX USB Storage
 target=0 lun=0     Direct-Access(0) - Generic  USB SD Reader    Rev: 1.00
 target=0 lun=1     Direct-Access(0) - Generic  USB CF Reader    Rev: 1.01

[1] + Done                  io-usb -v -duhci
# devb-umass cam pnp verbose
# ls /dev/hd*
/dev/hd0       /dev/hd0t79    /dev/hd1       /dev/hd2
# mount -t qnx4 /dev/hd2 /mnt/flash
# ls /mnt/flash/
./                .altboot          .boot             .longfilenames
../               .bitmap           .inodes
#
```

Printout 4.6: Mounting CF card on QNX.

```
# fdisk /dev/hd2
# dinit -h -f velo_v2.ifs /dev/hd2
DINIT: You have specified the 'raw' disk, not a partition: continue (y/n) ? y
Using loader /usr/qnx630/target/qnx6/x86/boot/sys/ipl-diskpc2-flop
Disk '/dev/hd2' contains 2015232 blocks (1007616K).
#
```

Printout 4.7: Loading the OS image into the flash memory.

```
Hit Esc for .altboot.....................................................
.....................................S
```

Printout 4.8: Message appearing on the CyberBikePC after the putting new OS image on the CF card.

```
# dinit -hb /dev/hd2
```

Printout 4.9: Solution suggested by Loftum [2006], but not sufficient this time.

Then several attempts of getting the CyberBikePC to boot from the flash card were done, including:

- searching the internet for similar problems and suggestions
- putting the old original image onto the CF card again
- using IDE-card-readers in the CDROM[8] slot on the QNX host workstation instead of the USB card-reader
- calling Loftum for help

These attempts ended in an assumption that the card reader were playing tricks with the boot sectors on the CF card. Loftum had used a different card reader, but the QNX workstation, most of the steps listed in his thesis and even the ifs-file were the same. Hence the card reader was causing the problems. If another card reader was available (and even better; the same as Loftum used) it should have been tried, in order to ensure that the source of the problem to be found. But after spending too much time trying to solve this problem, alternative solutions became more attractive. This included installation of a mobile hard drive, and is described in Section 5.1.

[8]Compact Disk Read Only Memory

4.3 Propulsion motor

4.3.1 Placement

A new propulsion motor had to be mounted on the bike. The intention was to place it where the old one was seated, as shown in picture Figure 4.5.

Figure 4.5: The CyberBike as it looked by the start of this work.

As it can be seen from Figure 4.5, the front mudguard is nearly touching the tachometer on the motor. The tachometer and its cables should be shielded from the front wheel in some way.

The solution to this was to mount the propulsion motor on the vertical tube. Another problem that got solved by moving the motor, was a conflict between the chain and the battery-frame. When the motor is located at the upside of the pedal gear, it will not lower the chain, and thus avoiding the conflict between the chain and battery frame, as shown in Figure 4.6.

The disadvantage by this solution is that the bike is getting an even higher center of gravity, making it less stable. Still, this change should not make a crucial difference in the bikes stability.

4.3.2 Connection to Baldor TFM 060-06-01-3

The motor was connected to the Baldor TFM 060-06-01-3, at the M1+ and M1- pins. This is documented in Table E.3 in Appendix E.

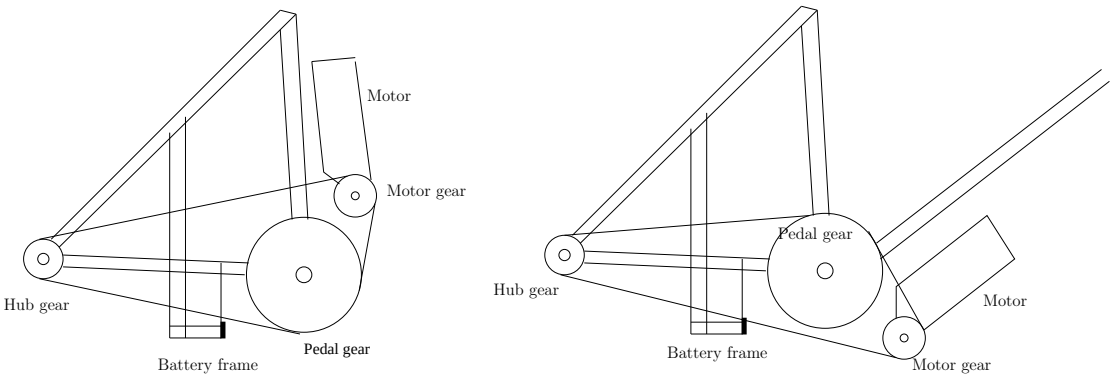

Figure 4.6: Left: Chain is not in conflict with battery frame (black part). Right: here we have a conflict.

When applying a voltage across the V+ and V- pins, there was no reaction on the motor, neither through software (see Printout 4.10) or through the I/O-card (DMM-32-AT), nor by simply connecting an external power source to the pins. This last approach eliminated any potentially errors in the system somewhere above[9] the motor controller card. But when disconnecting the motor from Baldor TFM 060-06-01-3, and power it up directly from a power source, it runs like expected. Hence the motor controller card had to be investigated.

```
# devc-dmm32at verbose &
[1] 577574
devc-dmm32at:    Verbose mode.
devc-dmm32at:    analog write, Inode 1
devc-dmm32at:    Writing voltage: 2.500000
devc-dmm32at:    Writing DMM value: 3071
DMM-32-AT:       Writing: 0xbff (3071) to channel 1


(In another terminal window:)
# cd /dev/dmm32at/analog/out
# echo 2.5 > da1
#
```

Printout 4.10: Setting the control voltage for the propulsion motor by software.

In [Baldor ASR, 1988] it says that all the three motor axis have their own set of potmeters, placed at the backside of the Baldor TFM 060-06-01-3:

1. Tacho voltage scaling

[9] To use the word "above" here, it is assumed that the whole system could be viewed hierarchical, with the motors and actuators at the bottom, then the hardware connected to these, and then the motherboard with its CPU before the software lies on top.

2. Velocity loop gain
3. Current limit
4. Offset

Tuning these finally made the motor start running.

4.4 GPS

For the CyberBike to know its absolute position, a GPS-module was suggested as a nice device. The bike is considered to be used for PR purposes for the Department of Engineering Cybernetics, and a GPS installed on a bike – which indeed is a bit uncommon – extends the impression that the bike is utilizing cutting edge technology. This gives room for some enhancements, like using the GPS measurements for comparing and adjusting the speed measurements, and navigation algorithms (see Section 8.3).

A *GlobalSat EM-411* GPS-module was ordered from [ELFA, 2007] for this purpose. Figure 4.7 shows this unit. This is the same module used by Eriksen [2007] and Bjørntvedt [2007] in their master's theses. Some benefits were drawn from the fact that three students at ITK were using similar GPS modules at the same time.

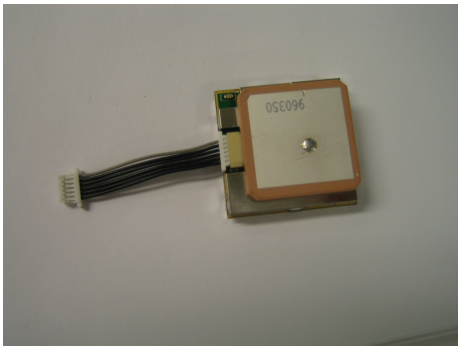

Figure 4.7: The EM-411 GPS module from GlobalSat

The various messages that could be received and sent to the GPS module are defined in [GlobalSat Technology Corporation]. Some specific data for the unit are given in Table 4.2.

4.4.1 PCB

Eriksen made a circuit diagram and PCB layout in Eagle, which was also used in the CyberBike project. The layout files are included in the appended CDROM, see Appendix A.

After etching and soldering the PCB, it turned out that the photoresist was not properly removed before the tin coat was applied to the board, but – due to the candidates

General	
Chipset	SiRF Star III
Frequency	L1, 1575.42 MHz
C/A code	1.023 MHz chip rate
Channels	20 channel all-in-view tracking
Sensitivity	-159 dBm
Accuracy	
Position	10 meters, 2D RMS
	5 meters, 2D RMS, WAAS enabled
Velocity	0.1 m/s
Time	$1\mu s$ synchronized to GPS time
Acquisition Time	
Reacquisition	0.1 sec., average
Hot start	1 sec., average
Warm start	38 sec., average
Cold start	42 sec., average
Dynamic Conditions	
Altitude	18,000 meters (60,000 feet) max
Velocity	515 meters /second (1000 knots) max
Acceleration	Less than 4g
Power	
Main power input	4.5V $\sim$ 6.5V DC input
Power consumption	60mA
Protocol	
Electrical level	TTL level, Output voltage level: 0V $\sim$ 2.85V
	RS-232 level
Baud rate	4,800 bps
Output messages	NMEA 0183 GGA, GSA, GSV, RMC, VTG, GLL

Table 4.2: Specific data for the GPS module from GlobalSat

litlle PCB creating experience – this was not discovered until the soldering was done. This resulted in a bad-looking soldering side of the board. The card was washed with methylated spirit, and applied a new tin coat, and the soldering points were reheated to improve the connections. Probing all connections on the board by the multimeter (see Section 6.1.1) showed that the connections were good, even though the board looks bad on that side. Figure 4.8 shows the soldering side of the board before and after the resoldering.

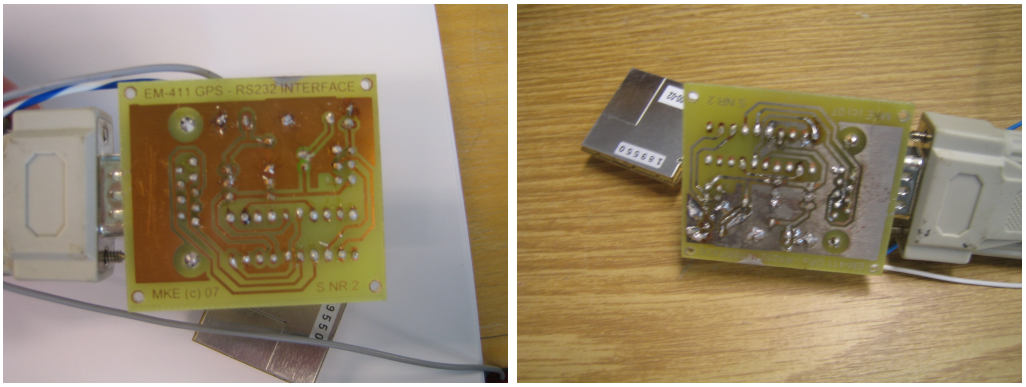

Figure 4.8: The soldering side of the GPS board. Left: before resoldering. Right: after.

The component side of the PCB looks somewhat better, and is shown in Figure 4.9. It could be seen from this figure that a voltage controller, a LED[10], a resistor, two capacitors, two headers, a DB-9 connector, and two screw terminals (for Vcc and Gnd) is soldered to the board. How the GPS-module is connected is also shown.

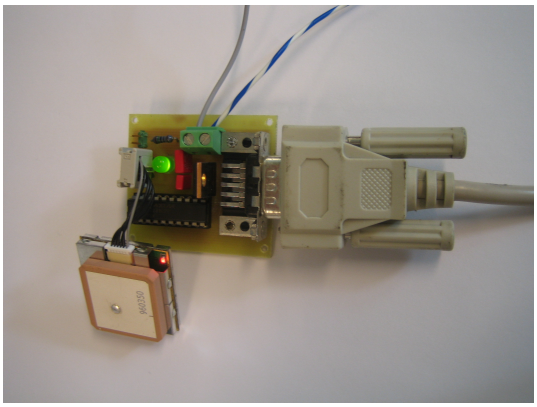

Figure 4.9: The component side of the GPS board.

[10]Ligth Emitting Diode

4.4.2 The devc-gps driver

The driver for the GPS was made as a resource manager, similar to the `devc-imu`, with a serial device reading thread, and a resource manager thread, handling client requests. When a gps-message is received, it is parsed by the functions in `gps-messages.c`, called by the serial device reading thread. There exist one parse function for each message that could be received. These functions stores the values received in local variables, until they by the end of each function locks a mutex before the parsed values are stored in a measurement struct. Code sample 4.7 shows the measurement struct, and Code sample 4.8 shows an example on the writing process to the measurement struct, taken from the *parse_ gpgll()* function.

Code sample 4.7 The type of the measurement struct, defined in `gps.h`

```
typedef struct meas {
  int     date;
  int     hour;
  int     min;
  int     numberofsatellites;
  int     satellites[12];
  double  sec;
  double  latitude;
  double  longitude;
  double  hdop;
  double  altitude;
  double  geoidseparation;
  double  pdop;
  double  vdop;
  double  speed;
  double  course;
} meas_t;
```

In `devc-gps.c` the functions *gps_ devctl()* and *gps_ read()* are defined. Both is locking the mutex the same way as shown in Code sample 4.8, before reading the requested data out of the measurement struct. Some messages could be sent to the GPS module, in order to configure and request specific data from the device. These messages are not used in the current version of the driver, because it was assumed that the CyberBike's GPS usage would be kept at a moderate level, and only the basic functionality provided as default was desired.

The GPS resource manager is registering its device files under the `/dev/gps/` directory, and when the driver is running, there will exist a file corresponding to each of the elements in the measurement struct in Code sample 4.7.

Code sample 4.8 Example from *parse_ gpgll()* on storing values in measurement struct.

```
// Lock mutex
if (pthread_mutex_lock(&val_mutex)){
  printf("%s:\tCouldn,t lock mutex.\n", NAME);
}
measurements.hour           = hour;
measurements.min            = min;
measurements.sec            = sec;
measurements.latitude       = latitude;
measurements.longitude      = longitude;

// Release mutex
if ( pthread_mutex_unlock(&val_mutex)){
  printf("%s:\tCouldn,t unlock mutex!\n", NAME);
}
```

4.5 Pendulum Limit Switches

This section describes the first step in implementing an angle limiting function on the inverted pendulum. The purpose is to avoid that the pendulum motor is giving a torque that would move the pendulum to a bigger angle, when it reaches its end positions. These end positions are given by the frame mounted around the lower part of the inverted pendulum, providing a physical obstruction when the pendulum tries to move too far away from its vertical position. On this frame two limit switches were mounted, but not connected to the motor in any way. In the following, a description of how these switces first were connected, and why this was wrong. For the busy reader, only interested in how the implementation ended up, refer to Section 5.5.

Loftum [2006] suggested a way to implement the functionality for the switches. The proposal included no software; only two diodes to disallow reference voltages into the Baldor TFM 060-06-01-3 that would make the pendulum move further away from its vertical position if a switch is trigged. The connection diagram is shown in Figure 4.10.

Different alternatives were evaluated. The first one evaluated was to connect the switches to the DMM-32-AT input ports, and implement the safety function in software. As soon as the signal from the switches could be observed by the software, additional functionality could be added, e.g. self-calibrating of the pendulum angle in a start-up procedure.

But it is generally a bad idea to mix the control and safety related functions toghether. As an example, consider the case where the control system stops as a consequence of a software error (segmentation fault, overflow, etc.), the safety function might not be executed, and the the last value written to the DMM-32-AT would remain on its output channels until it is powered down.

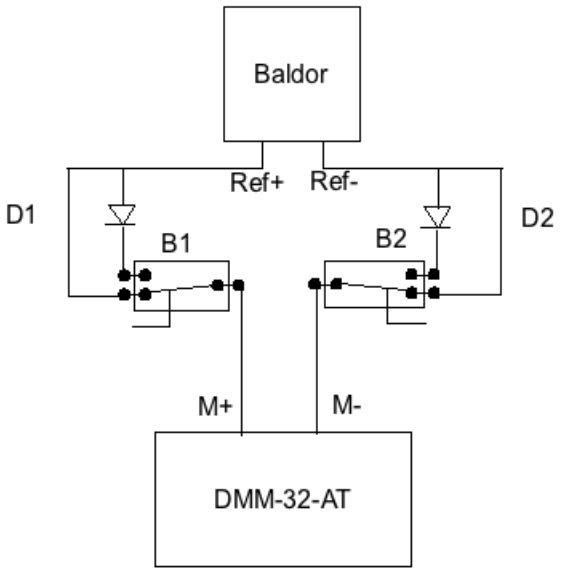

Figure 4.10: Loftum's [2006] connection proposal for the pendulum limit switches.

Loftum [2006] also argues that the proposed hardware solution does not consume any CPU. That is true, but with an interrupt based solution, the processing power used for this function would be minimal during normal operation. This is based on the assumption that the pendulum angle is kept small when the CyberBike is operating under normal and stable conditions, and that the limit switches are to be used only when the system is about to fail.

Figure 4.11 shows a circuit diagram of how the function was implemented at first. It is the same system as shown in Figure 4.10, but with some added information on how it is connected via the terminal block (X1), and which pins are connected where.

Even if the diodes is providing the pendulum angle limiting function, the signal from the switches may still be connected to the input ports of the DMM-32-AT to provide information about switch activation to the control system software, if it is desired. Some capacitators to avoid transients when the switch is trigged is probably wise. Even if the switches are providing a high or low signal, it should not be connected to the digital I/O channels on DMM-32-AT without some current limiting circuitry or decoupling mechanism. The D/A output channels (which is connected to the switches other end), is capable of delivering more current than the digital I/O can handle.

4.5.1 Testing and debugging

The soluton did not work as expected for the right switch, B2. A lot of probing and testing was done to point out where the error was. This appeared to be more difficult

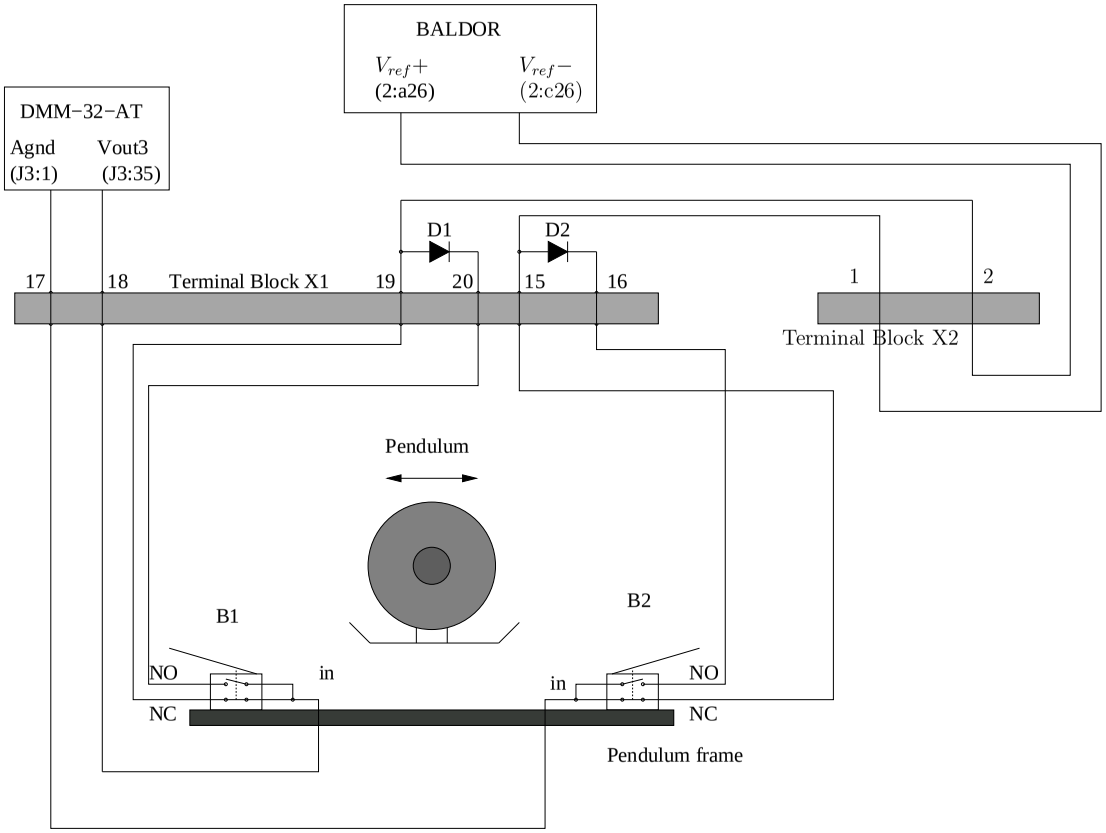

Figure 4.11: Circuit diagram of first connecton of the pendulum limit switches. Terminal block connections are indicated.

than expected. No wrong connections according to Figure 4.10 and Figure 4.11 could be detected.

It is worth noting that pin 1 on DMM-32-AT, *Agnd*, was connected to the input pins on B2, after the connection was forked off to supply the negative reference signal to the other two motor axis. If connected otherwise, all the other motors would also stop when the B2-switch was activated.

Setting a voltage out to the pendulum motor had an impact on the other motors, which indeed is undesirable. When the propulsion motor was running at a low speed, it stopped when activating B2. Also, a voltage across the negative input references on Baldor TFM 060-06-01-3 was observed when B2 was closed.

Then it was realized that cutting the ground connection from DMM-32-AT to the motor controller card – as done in D2 – is not the same as setting the reference voltage to zero. The internal circuitry of Baldor TFM 060-06-01-3 was playing a role in this case, as well as the exact timing of when B2 cut the signal out on its normally closed (NC) output pin and connects the normally open (NO) output pin to its input pin. To fully understand what went wrong, a deep investigation of both devices should be made.

This investigation was not done. Instead it was assumed that cutting off the ground connection between DMM-32-AT and Baldor TFM 060-06-01-3 was the mistake done here, and an alternative circuit was designed to correct it. This solved the problem. Section 5.5 describes this solution. A note on such failure situations and debugging is made in Section 7.4.

Chapter 5

Solution

5.1 OS and storage upgrade

Some problems occured during the upgrade of the OS image on the CF memory, probably related to the card reader that was used. This is described in detail in Section 4.2.

5.1.1 Installation of hard drive

An ordinary Fujitsu MHK2060AT [Fujitsu, 1999] hard disk drive was mounted to the EIDE-slot on the Wafer-9371A. The disk was an older mobile 2.5 inch 44 pins ATA[1] with approximately 6GB storage space. This is not very much for a new laptop, but for the CyberBikePC it should be more than sufficient. The reasons for choosing this disk were simple. It had already done its duty on a robot developed at ITK participating in an *Eurobot* competition some years ago, and the disk was now unmounted and not used anymore, hence a very cheap alternative for this project.

At this point, the Fujitsu FMHK2060AT mobile hard disk was plugged into the QNX stationary host workstation instead of its current disk drive, and a QNX Neutrino RTOS 6.3.0 SP3 CD[2] were put in its CDROM station. The IDE-slot at the workstation got 40 pins, hence the 4 pins for powering up the disk needed to be connected elsewhere. A simple converter with a Molex plug for power was used for this purpose, see Figure 5.2.

QNX was then installed at the hard disk, as described in [QSSL, 2005, chap. 3]. At the point where the installation procedure wanted to reboot the system for the first time, the PC was entirely shut off instead of rebooted, and the disk was unplugged from the PC and put into the CyberBikePC. This was done because QSSL [2005] stated that

> *"After rebooting, your hardware will automatically be detected."*

and because it was known that both the QNX host PC and the CyberBikePCwere x86 intel based computers, and hence they could possibly use the same basic parts of the

[1] Advance Technology Attachment
[2] Compact Disk

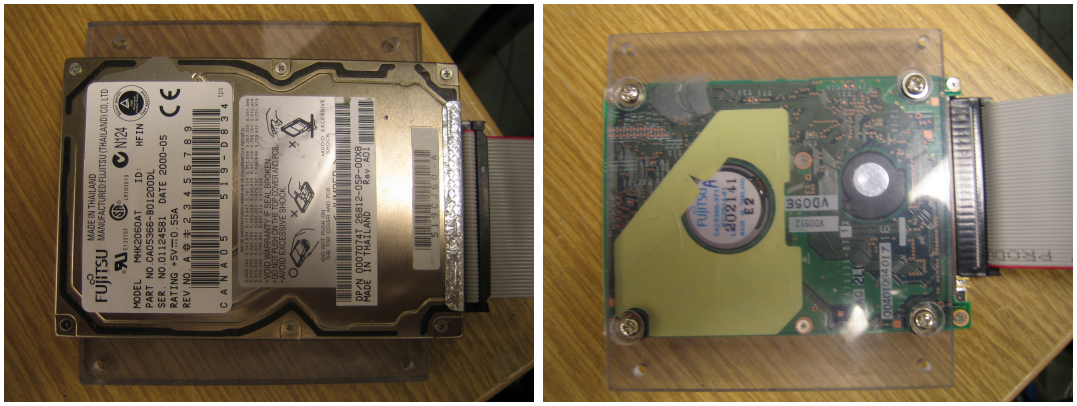

Figure 5.1: The Fujitsu MHK2060AT hard drive shown both from its over and under side. A piece of plexi glass is glued to the underside of the disk to ease mounting.

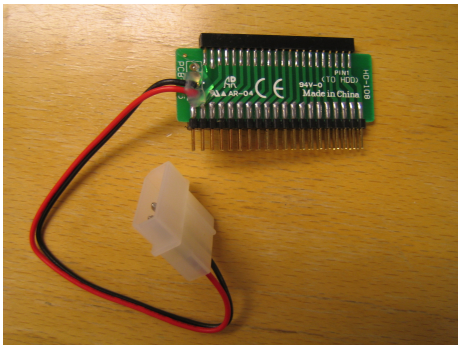

Figure 5.2: 40 to 44 pins IDE connector converter.

installation. When the CyberBikePC was powered up, its hardware was detected, and a fully working QNX installation were running on the bike's baggage rack computer.

The disk was finally mounted at the bottom plate in the suitcase, below the Baldor TFM 060-06-01-3, to utilize space.

5.2 MTi

The MTi was connected to the Wafer by a serial cable into COM1. A protective housing for the device was made to put the device in. This is shown in Figure 5.3. It was aimed at mounting it as low as possible, in order to keep it away from the shaking from the pendulum, and to be close to the position where the roll angle is defined in the control system. Ideally, the MTi should have been placed at the point where the rear wheel is touching the ground, but obviously this is hard to implement in real life.

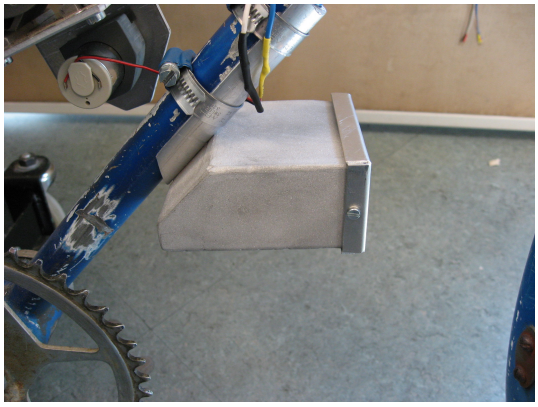

Figure 5.3: The MTi protective housing.

5.2.1 Driver

The software driver for the MTi was made as a resource manager (see Section 2.2.2), based on the code for the Spark Fun IMU made by Loftum [2006], and the example code delivered with the kit from Xsens. The latter was written in C++, making it a natural choice to use that language for the rest of the driver. It had small impact on the code in this case.

First the example code was compiled in QNX, to make sure it really was compatible. The only change that had to be made was in `MTComm.cpp`, where a non-POSIX command had been used, see codesample Code sample 5.1. The solution to the compile error was found by searching on [OpenQNX].

A new C++ project was made in the QNX Momentics IDE, called `devc-mt`, before the files `MTComm.cpp` and `MTComm.h` was copied into it (with the changes shown in Code sample 5.1). Then the code from `devc-imu.c` and from `mt-example.cc` (from the

Code sample 5.1 The only change made in `MTComm.cpp` to make it compile in QNX.

```
    // Disable hardware flow control
    options.c_cflag &= ~CRTSCTS;

Changed to:

    // Disable hardware flow control
    options.c_flag &= ~IHFLOW;
    options.c_flag &= ~IHFLOW;
```

vendor) was merged into the master driver file `devc-mt.cc` to reuse some already tested and developed resource manager code, and at the same time make use of the example calls to configure the MTi-communication. The resource manager was built around the same structure as in the Spark Fun IMU case. Two threads were used: waiting for requests by the *dispatch_ block()* function call, and the other reading data from the serial device, storing it in a shared measurement array. This array is protected by a mutual exclusive semaphore mechanism, provided by POSIX.

The word "merge" used above is – strictly speaking – a simplification. This operation included some rewritings, variable and function name changes (e.g. from *imu_ read()* to *mt_ read()* etc.), type changes, debugging and testing to get things correct.

Usage

The example code requested user inputs during initialization, to set up the correct serial device, if calibration and/or orientation data should be presented on the screen, and if the orientation data should be given as rotation matrices, quaternions, or Euler angles.

These settings could now be set from the command line, as arguments to the driver, insted of via *scanf()* function calls during initialization. A usage example of the driver is shown in Printout 5.1.

```
# devc-mt -vv -m3 -f2 -s /dev/ser2
```

Printout 5.1: Usage example of `devc-mt`.

This would start the driver in noisy mode, showing both calibration and orientation data, using the COM2 serial port. Starting the driver in verbose mode (`-v`), will only show debug related information, and not the measurements. A usefile is compiled into the driver, enabling the `use devc-mt` command to give an explanation of how the driver is started with its various options, similar to other QNX programs.

The CyberBike is using the default settings for the driver, which is both calibration and orientation data, the latter given as Euler angles. Using Euler angles yield some singularities when the pitch angle is reaching 90°, which makes quaternions or rotation matrices a better choice for some users. This is not a problem in this project, and Euler angles is chosen to integrate with the controller made by Bjermeland [2006].

Output from the resource manager is provided through the device files registered under the `/dev/mt` directory. In default mode, this includes the files:

- `/dev/mt/calib/acc/x` holding acceleration in x direction $[m/s^2]$
- `/dev/mt/calib/acc/y` holding acceleration in y direction
- `/dev/mt/calib/acc/z` holding acceleration in z direction
- `/dev/mt/calib/gyr/x` holding rate of turn in x direction $[rad/s]$
- `/dev/mt/calib/gyr/y` holding rate of turn in y direction
- `/dev/mt/calib/gyr/x` holding rate of turn in x direction
- `/dev/mt/calib/mag/x` holding the magnetic field in arbitrary units in x direction, normalized to earth field strength
- `/dev/mt/calib/mag/y` similar in y direction as for the x
- `/dev/mt/calib/mag/z` as for x and y direction
- `/dev/mt/orientation/roll` the roll angle (ϕ) measured in degrees, between $-180°$ and 180°
- `/dev/mt/orientation/pitch` the pitch angle (θ)
- `/dev/mt/orientation/yaw` holds the heading angle (ψ)
- `/dev/mt/samplecounter` a timestamp put in the message from the MTi device

If another orientation output mode is chosen, the device files would have corresponding names, like `orientation/q1` through `orientation/q4` in quaternion mode, or all letters from `orientation/`a to `orientation/i` in matrix mode. The latter is defined as shown in Equation (5.1).

$$R = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \tag{5.1}$$

The *mt_ read* and *mt_ devctl* functions are made to update the content in these files. No *mt_ write* funtion is implemented by this resource manager. A large set of messages defined by the MT-protocol for configuring various aspects of the device [see Xsens,

2005, p. 33], to be sent from the host to the devcie, and hence a write function could be useful to control its behaviour from other programs (like `devc-velo`. But to keep the system complexity at a moderate level, it was decided that this was unnecessary. If such configurations is wanted in the future, this probably could be done throug hard-coding it in the MTi driver, or add some functionality in the `MT_DEVCTL_SETVAL` block in the *mt_ devctl()* function.

5.3 Propulsion motor

In the CyberBike model by [Bjermeland, 2006], the bike velocity is not controlled by feedback of any measurements in software. Also, the model and controller is designed for being valid for a constant speed and on a level surface, leading to less demanding control functions for this motor. A simple controller (i.e. PID[3]) in software would probably have done the job, but using the available processing power a bit sparingly is preferred, and therefore it was decided to go for other alternatives instead.

The Baldor TFM 060-06-01-3 is capable of performing speed control, by altering the jumper settings for the particular axis, and connect a tachometer measurement back to the controller. The propulsion tachometer's positive signal was connected to the motor controller card's pins 2:c4 (via terminal 1 on block X1) and the negative signal was connected into 2:c14 (via X1:12).

To get some velocity measurement data in the software running part of the control system, it was desired to keep the tachometer connected to the DMM-32-AT. But using the existing setup for the propulsion tachometer into A/D channel 0, would make the tachometer be connected to Agnd on the I/O card, which in turn is connected to the negative reference voltage on pins 2:c2, 2:a26 and 1:a16, at the same time as it is connected to "Ref gnd" on 2:c14 at the Baldor card. These pins seems to be unconnected inside the card [Baldor ASR, 1988].

DMM-32-AT is set up to have 8 differential input channels, previously unused. The propulsion tachometer was connected into one of these, channel 8, to avoid any ground loop problems. The `devc-velo` driver was updated to use the `ad8` as its velocity measurement device.

The propulsion motor controller is tuned by four potmeters at the Baldor TFM 060-06-01-3's front side (at the very back of the bicycle). The parameters tuned by these potmeters are described in Section 4.3. Due to some problems with the DMM-32-AT output channels (see Section 6.5), the tuning of the motor was not applied very successfully. A small offset seems to appear sometimes, as the components in the system gets warmer, and the motor tend to be rotating with *very* low speed even if the reference voltage is set to zero.

A proper tuning of the bikes velocity controller is left undone, as reasonably good velocity control seems to be unattainable as long as the DMM-32-AT is giving unstable output.

[3] Proportional-Integral-Derivative (controller)

5.4 New driver for DMM-32-AT

During the work on this thesis, some problems with the CyberBike's I/O card was discovered. Section 6.5 describes this process in detail. Several errors was discovered. Both in hardware and software. When searching for ready-made download software from the vendor as an attempt to eliminate the `devc-dmm32at` resource manager as an error source, it was discovered that Diamond Systems Corporation [2007] provided a universal driver-library for QNX systems.

Why this library wasn't used in the first place is unknown. Studying the *dmm32at_ analog_ write()* implementation in `dmm32at.c` compared to the description of how to perform a D/A conversion in [Diamond Systems Corporation, 2003, p. 36], showed that it probably would have been wise. This erroneous code have been used on the bike for a long time. It might have caused the hardware errors, but it is neither explored, nor discussed further in this thesis.

Changing the code to use the *uscud* (Diamond Systems Corporation's Universal Driver), affected mosly the `dmm32at.c` file. The resource manager and thread setup was kept as it was. The `libdscud5.a` had to be copied into a place where the QNX Momentics IDE's compiler could find it. In the case for the installation on the Windows XP PC used for development on the CyberBike, the library was placed in `C:\QNX630\target\qnx6\usr\include\dscud5\`, and *dscud5* and *m* (for the math-library) was added to the *LIBS* variable in the `devc-dmm32at` projects makefile, `common.mk`.

How to use the universal driver is described thoroughly in [Diamond Systems Corporation, 2007], by API and examples. The core of the code for the new read and write functions are shown in Code sample 5.2. The actual code performs error handling, initialization and resetting of global variables, scaling of the return value (in the read case), and some printouts's if in verbose mode, not shown here.

The library from Diamond Systems provided functionality for auto calibration, which was at first put in the initialization function for the driver. But because of the hardware errors, described in Section 6.5, this had to be put in a separate function; *dmm32at_ autocal()*. An option to be used from the command line when starting the driver was then added to make use of the autocalibration utility. This should only be used with the da0 channel disconnected from its pin on the DMM-32-AT's J3 header, because the result of the autocalibration seems to be depending on the resistance on this pin. This should not be a problem if the I/O card gets repaired. The autocal functionality is demonstrated in Printout 5.2.

5.5 Pendulum limit switch - final implementation

The limit switches was finally connected like shown in Figure 5.4 (schematic overview), and Figure 5.5 (detailed; with terminal block connections indicated).

The most positive effect of the installation is that the motor will not be powered to try to push the pendel any further when it reaches its end position (the frame). This prevents

Code sample 5.2 Core of the new *dmm32at_analog_read()* and *dmm32at_analog_read()* functions.

```
int dmm32at_analog_read(double *valptr, int channel){

  // Step 1: Not shown, includes some initialization
  // of the A/D setting structure, depending on if
  // it is a tachometer (+/- 10V) or a potmeter (0-5V)

  // Step 2: setup the driver for A/D operations:
  dscADSetSettings(dscb, &dscadsettings);

  // Step 3: perform A/D conversioin and get sample:
  dscADSample(dscb, &dscsample);

  adval = (int) dscsample;

  // Step 4: not shown, includes scaling of return value,
  // to a double value between -5 and +5
  // (reused code from previous dmm32at-driver).

}

int dmm32at_analog_write(DSCDACODE val, BYTE channel){
  dscDAConvert(dscb, channel, val);
  return 0;
}
```

```
# ./devc-dmm32at verbose autocal
devc-dmm32at:   Verbose mode.
devc-dmm32at:   Autocalibration will be performed during init.
devc-dmm32at:   If DMM-32-AT-card still is buggy, you should not do this while p
in 38 (da0) on J3 is connected to anything.
DMM-32-AT:      Performing D/A-auto calibration...
DMM-32-AT:      Done D/A-autocalibration. Offset Error:    -0.500, Gain Error:
   0.350
DMM-32-AT:      Performing A/D-auto calibration...
DMM-32-AT:      Configuration Mode: 0, Offset Error:     0.180, Gain Error:     0.316
DMM-32-AT:      Values for offset and gain met specified tolerance
DMM-32-AT:      Configuration Mode: 1, Offset Error:     0.450, Gain Error:     0.292
DMM-32-AT:      Values for offset and gain met specified tolerance
DMM-32-AT:      Configuration Mode: 2, Offset Error:     0.080, Gain Error:     0.424
DMM-32-AT:      Values for offset and gain met specified tolerance
DMM-32-AT:      Configuration Mode: 3, Offset Error:     0.320, Gain Error:     0.128
DMM-32-AT:      Values for offset and gain met specified tolerance
DMM-32-AT:      Configuration Mode: 8, Offset Error:     0.430, Gain Error:     0.028
DMM-32-AT:      Values for offset and gain met specified tolerance
DMM-32-AT:      Configuration Mode: 9, Offset Error:     0.270, Gain Error:     0.084
DMM-32-AT:      Values for offset and gain met specified tolerance
DMM-32-AT:      Configuration Mode: 10, Offset Error:     0.420, Gain Error:     0.018
DMM-32-AT:      Values for offset and gain met specified tolerance
DMM-32-AT:      Configuration Mode: 11, Offset Error:     0.020, Gain Error:     0.446
DMM-32-AT:      Values for offset and gain met specified tolerance
DMM-32-AT:      Configuration Mode: 12, Offset Error:     0.724, Gain Error:     0.000
DMM-32-AT:      Values for offset and gain met specified tolerance
DMM-32-AT:      Configuration Mode: 13, Offset Error:     0.022, Gain Error:     0.000
DMM-32-AT:      Values for offset and gain met specified tolerance
DMM-32-AT:      Configuration Mode: 14, Offset Error:     0.456, Gain Error:     0.000
DMM-32-AT:      Values for offset and gain met specified tolerance
DMM-32-AT:      Configuration Mode: 15, Offset Error:     0.052, Gain Error:     0.000
DMM-32-AT:      Values for offset and gain met specified tolerance
#
```

Printout 5.2: Usage example of new DMM-32-AT driver, running autocalibration.

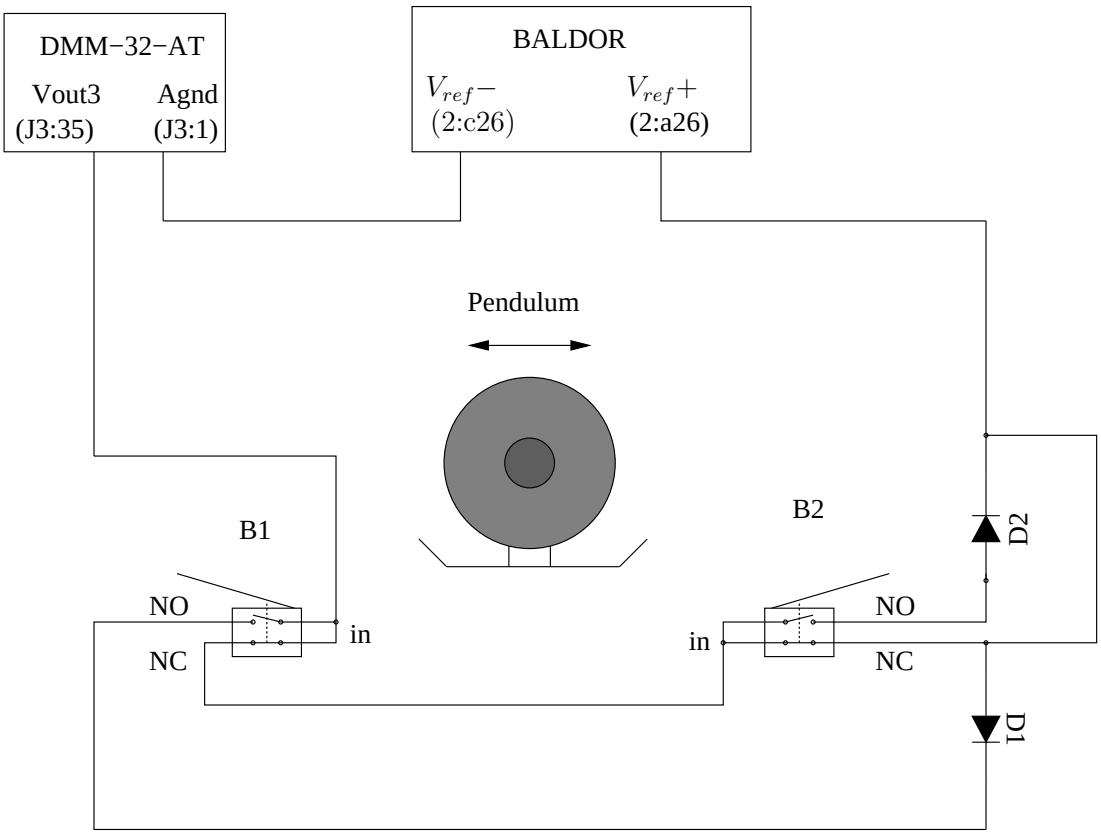

Figure 5.4: Overview of the pendulum limit switches' final connection circuit diagram.

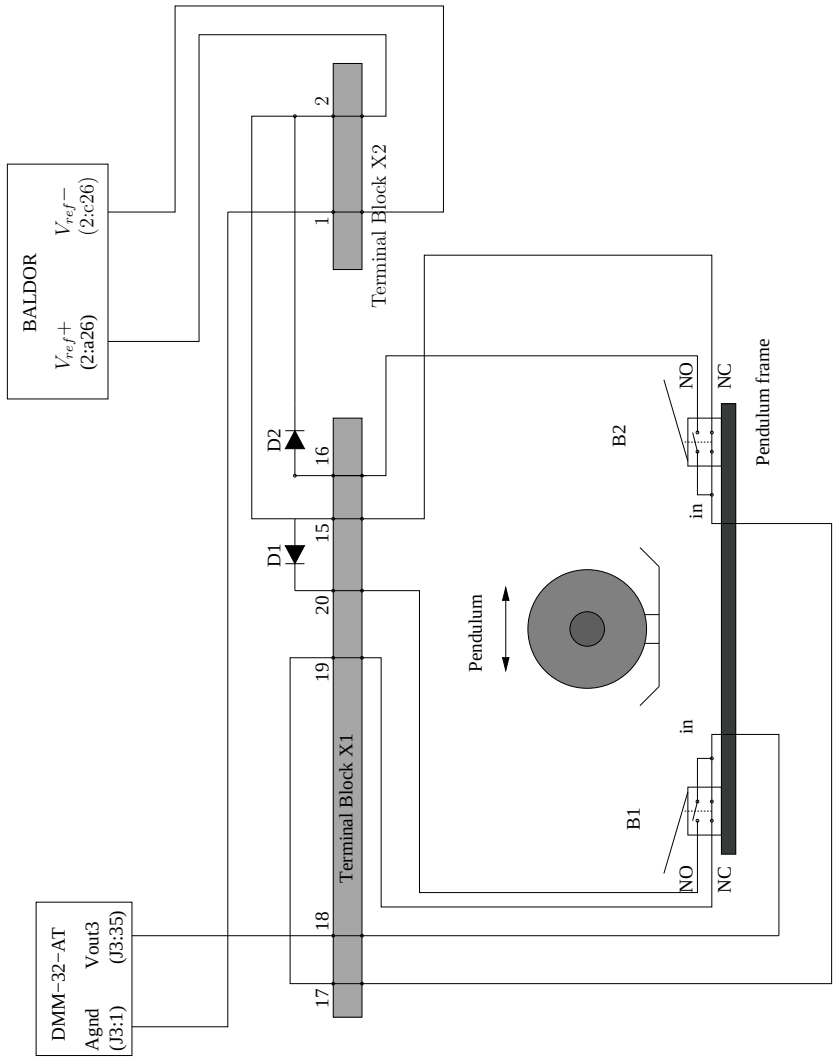

Figure 5.5: Circuit diagram of the pendulum limit switch final connection, with terminal block connections indicated.

destroying the motor brushes, which is a problem for DC-motors in static operation.

When the pendulum motor is moving towards its endpoint, it will still fall against the pendulum frame due to gravity and the speed it had when it reached the switch. This was not an unexpected nor unintelligible result. If the Baldor TFM 060-06-01-3 had been set up to control the *speed* of the pendulum motor instead of its torque, the behaviour would probably be different. The frame gets some rough treatment – less than before – but still it could be damaging to the equipment. Maybe some rubber knobs to absorb some of the momentum when the pendulum hits the frame would improve this, but this have not been tested.

In Section 4.5, connection of the switch signal into the DMM-32-AT was discussed as an option. The reason for this, was that it might could be used in the control system for detecting where the pendulum actually is, and to calibrate the pendulum potmeter value from this signal in a startup procedure. This is *not implemented*. The most important reasons is:

- The switching mechanism is assumed to give a less correct impression of where the pendulum actually is than the potmeters for the particular angles. The knob, mounted on the pendulum bar to push the switches, covers a relatively large area, which makes the switches closed for a wide range of angles.
- It is desired to keep the connection complexity in the CyberBike's suitcase as low as possible, to ease maintenance and debugging.
- No particular need for the information from the switches were detected in the control system. But it should be noted that the work in this thesis does not include sufficient amount of testing and tweaking of the control algorithm, to claim that the pendulum angle calibration functionality is not needed.

5.6 Emergency Stop Button

The 3 DC motors have the potential to being dangerous to people around. In an industrial environment such moving devices as rotating gears with chains, and swinging bars have to be put inside boxes or behind covers. The CyberBike is not equipped with many safety related devices and installations. Before serious testing is done, at least an emergency stop button should be mounted.

A button, switch, and a front adapter for mounting was ordered from [ELFA, 2007], and is shown in Figure 5.6. The switch-part is of type NC[4].

From [Baldor ASR, 1988, p. 1] it could be found that:

> *"It is possible to switch off the power stage on each axis via the disable inputs. The motor is without torque in this condition."*

Figure 5.6: The emergency stop button and switch. Picture taken from [EAO, 2007].

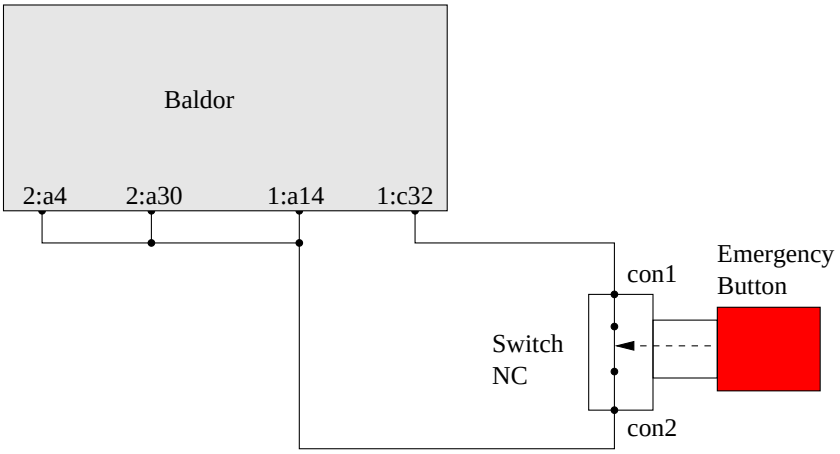

Figure 5.7: Circuit diagram of how the emergency button is connected.

And from that, the emergency switch was connected as shown in Figure 5.7.

The button was mounted at the back of the suicase, to let the operator stay *behind* the bike, away from the moving pendulum and handlebars, etc. Testing of this safety function worked as expected, and has shown to be useful in many cases. For instance; this button should be activated while starting up the system when the chain, connecting the propulsion motor to the rear wheel is installed. Else the bike might be moving a due to small disturbances during startup. The propulsion motor seems to be the most unstable at this point, because the speed controller on the motor controller card have some small stationary errors, fluctuating with the various components' temperature.

5.7 Fan

To give cooling to the PC and other boards and components, a fan was mounted in the CyberBike suitcase wall. This installation is particularly useful during tests of the bike, when the suitcase should be closed. An old and well-worn 12V fan was an inexpensive and easy solution, see Figure 5.8.

To be able to get the bottom plate up from the the suitcase, which is useful to access for instance the CF card slot under the Wafer-9371A, a part of the bottom plate was cut out to make it pass the fan.

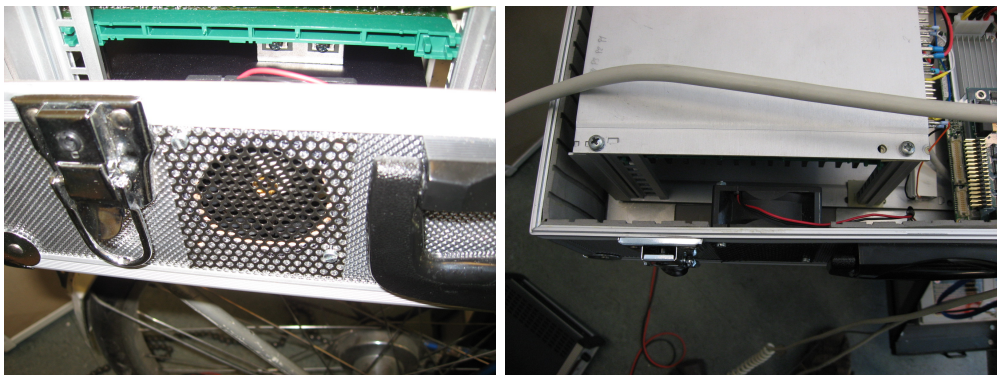

Figure 5.8: The cooling fan mounted at the CyberBike's suitcase.

It turned out that the fan was a bit noisy, and sitting beside it while developing and testing was a annoying. To solve this, a switch, shown in Figure 5.9, was mounted to cut the power when the suitcase is open.

5.8 Batteries

During the development process, a power source from Farnell (shown in Figure 6.3) was used, but for activities which includes forward motion of the bike, two sealed lead acid

[4]Normally Closed

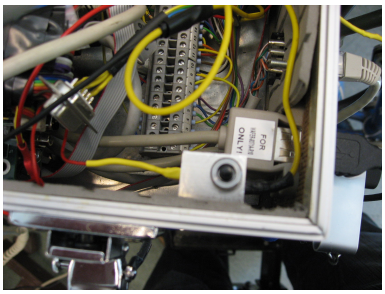

Figure 5.9: The fan switch mounted to be activated when suitcase is closed.

batteries from [ACT Batteries] was ordered. It was important that the batteries were either sealed, or were using gel instead of acid. There is no guarantee that the bike would stay on its wheels on every test run. Battery acid leaks has to be avoided. The sealed batteries seemed to be less expensive than the gel-batteries, and gives about the same amount of safety.

A tradeoff between size, weight, cost and amount of ampere-hours had to be made. The ATU12-35 model was chosen. The specification for these batteries are given in Table 5.1.

Variable	Size	unit
Nominal voltage	12	V
Rated Capacity	35	Ah
Length	195	mm
Width	130	mm
Case height	154	mm
Total height including terminal	184	mm
Approximate weight	10.5	Kg

Table 5.1: Specific data for the ATU12-35 batteries, from [ACT Batteries].

The batteries were placed in their dedicated frame, on each side of the bike. Cable shoes that were big enough for the batteries terminal screws were pinched to three wires, making the connection that can be seen in Figure 5.10. Two fuse holders with 15A fuses were soldered on to the wire close to the positive terminals, to provide protection from short circuiting and similar events. The right battery and its fuse are shown in Figure 5.11.

Now a switch is useful to be able to power down the system, without disconnecting any wires. This is mounted on the suitcase's foremost wall, and the positive wire from the batteries are cut connected via this switch, as seen from the connection diagram in Figure 5.10. The switch is shown in Figure 5.12.

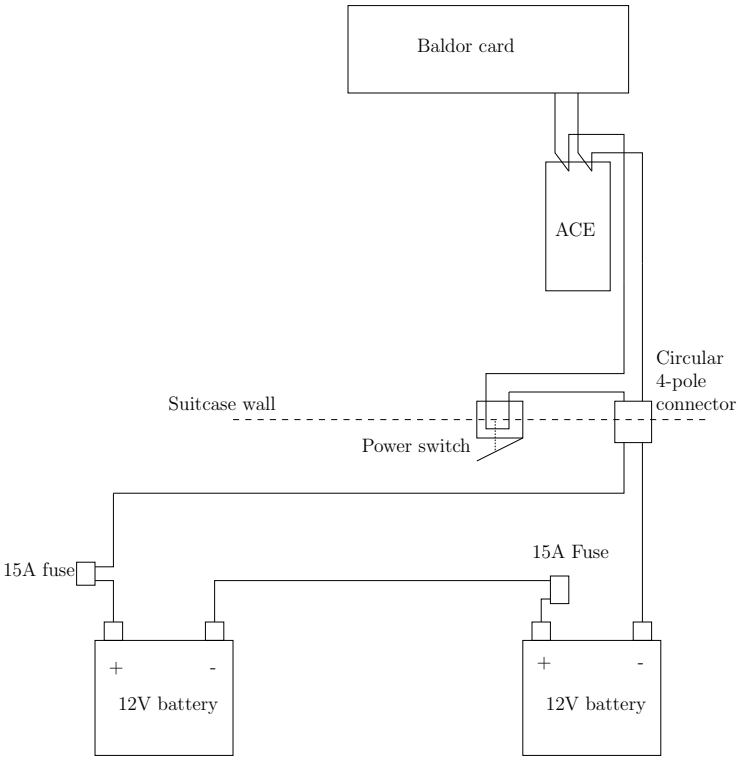

Figure 5.10: Connection diagram for the batteries.

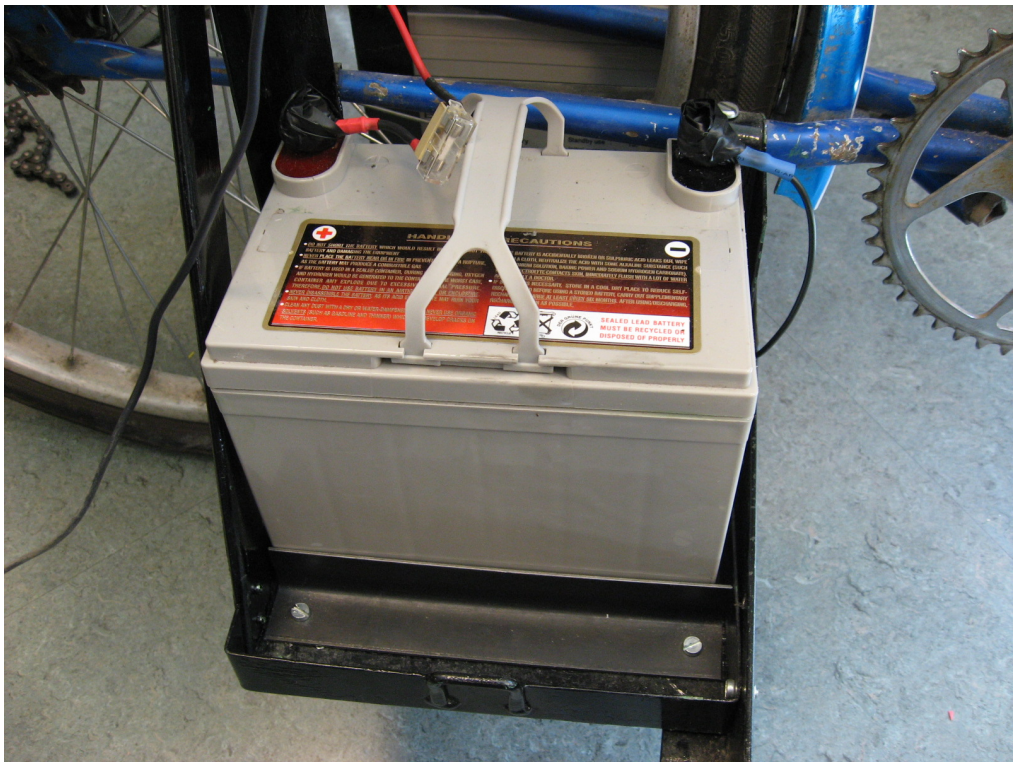

Figure 5.11: 12V battery from [ACT Batteries]. The 15A fuse and its fuse holder is soldered to the wire, close to the positive battery pole.

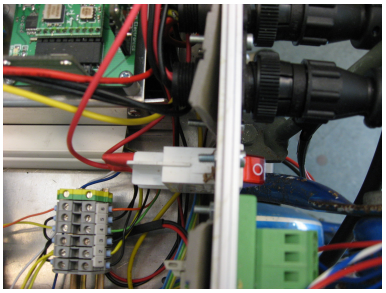

Figure 5.12: The power switch.

5.9 Driver summary

To compare the state of the CyberBike's software by the start and the end of this work, a figure comparable to Figure 3.13 is used to give an overview.

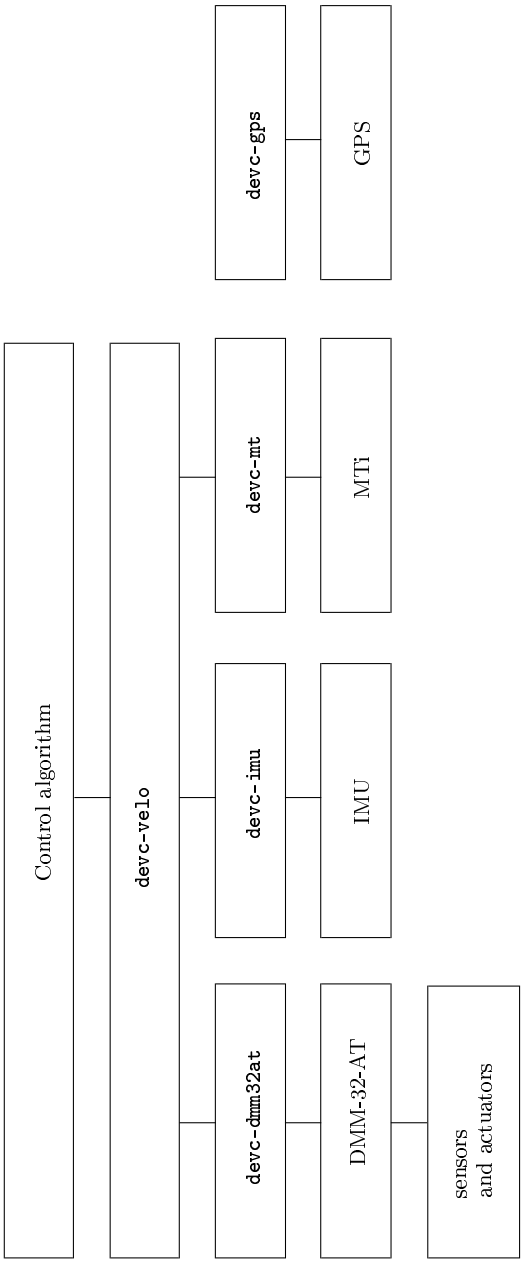

Figure 5.13: Schematic overview of the CyberBike's drivers by the end of the work on this thesis.

Chapter 6

Tests and experiments

6.1 Test equipment

6.1.1 Multimeter

The multimeter used during the work on this thesis is an "Escort EDM 168" with serial number 11116249, see Figure 6.1. The multimeter has got the most usual functions expected from such a device, including voltmeter, amperemeter, ohmmeter, DC and AC[1] setting, capacitance measurements, connectivity check ("beeping"), etc.

Figure 6.1: Escort EDM 168 Multimeter

6.1.2 Oscilloscope

An oscilloscope of type "Hameg HM507 Analog Digital Scope" (see Figure 6.2) with the serial number 021540457 was used in cases where the multimeter was inadequate, e.g. to check if a RS-232 signal could be detected on a wire.

[1] Alternating current

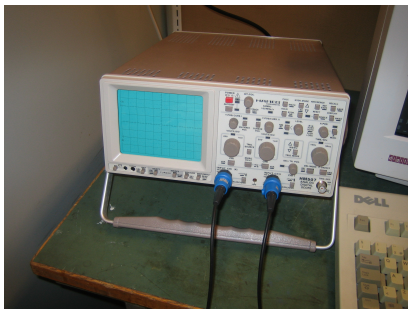

Figure 6.2: Hameg HM507 oscilloscope

6.1.3 Power sources

Two power sources were used; a "Farnell LS-30 Autoranging Power Supply" and a "TTi EL302T Triple Power Supply" with serial numbers 000666 and 256134 respectively. Both are shown in Figure 6.3.

Figure 6.3: Power sources. Left: Farnell LS-30. Right: TTi EL302T.

6.2 The pendulum motor

Fossum [2006, p 31] did not run a test drive on the pendulum motor in a "decent way"[2] early in his project, because the motor was mounted too elaborately. But he states that the motor have been connected to a power source and tested in both directions, to see that it worked. Later he told that the motor was having trouble lifting the pendulum when the $2kg$ weight was placed in its outermost position, and the pendulum was lying down to its side.

Therefore it was decided to unmount the pendulum motor to get a real test of it, without the pendulum. The way it was mounted made the work somewhat complicated,

[2]His own words in Norwegian was *"på en skikkelig måte"*

and the pendulum had to be removed from the motor after both were unmounted from the bike.

Without load, the motor seemed to work fine when applying $24V$ to its terminals. Further, the idea was to put the motor in a vise, and use a torque wrench on the axis. But the risk of damaging the motor by allowing too much current to flow through it while blocking the armature from rotating, made another approach more attractive. The alternative that came up was to tie a rope with a weight at its end on to the axis, while the motor was set in the vise. An approximation of how heavy the load would be, had to be calculated.

The weight m_w is $2kg$, and have a length of about $9cm$ and is placed at the end of the lever (pendulum), see Figure 6.4. The lenght of the lever l_p is approx. $0.62m$, hence

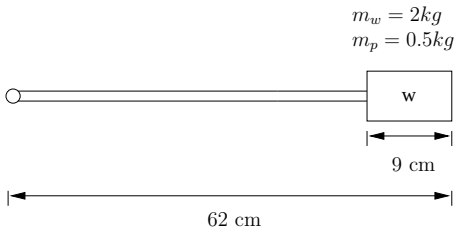

Figure 6.4: Sketch of the horizontal pendulum with weight at the end.

the weight's center of mass $x_{cm,w}$ would be about $0.57m$, which is the distance from the rotating center of the pendulum. The lever itself, m_p is about $0.5kg$, and would have its center of mass $x_{cm,p}$ at the middle of the lever, that is $0.31m$. The center of mass of the two items can be derived by:

$$\begin{aligned} x_{cm} &= x_{cm,p} + \frac{(x_{cm,w} - x_{cm,p}) \cdot m_w}{m_w + m_p} \\ &= 0.31m + \frac{(0.57m - 0.31m) \cdot 2kg}{2kg + 0.5kg} \\ &= 0.52m \end{aligned} \tag{6.1}$$

A mass of $2.5kg$ on a lever $0.52m$ from the rotating axis gives the torque:

$$\begin{aligned} T &= x_{cm} \cdot F_g \\ &= x_{cm} \cdot (m_w + m_p)g \\ &= 0.52m \cdot 2.5kg \cdot 9.81m/s^2 \\ &\approx 12.8Nm \end{aligned} \tag{6.2}$$

which the motor have to deliver as a minimum to raise the pendulum. In the case with a rope with a weight at the end tied to the motor axis, the weight corresponding to

the pendulum would be:

$$m_{corresponding} = \frac{T}{g \cdot r_{axis}} \tag{6.3}$$

$$= \frac{12.8Nm}{9.81m/s^2 \cdot 0.01m}$$

$$= 130kg \tag{6.4}$$

approximating the axis radius to $1cm$. The inaccurate measurements (the axis radius in particular) will have a large impact on this calculation, however; the calculations show that with that load amount, this is not the way to do the motor test.

Therefore the pendulum was mounted back to the motor. Then the motor could be tested with the actual load. The test showed no problems lifting the pendulum with the 2 kg weight placed at the end, from horizontal to vertical position. The problem is residing somewhere else.

The Farnell power source (see Section 6.1.3) was tuned to give a voltage and a current at approximately 24 V and 3.5 A respectively (close to the nominal values for the motors; 24V and 4A). When the current was reduced to 1 A, the motor could not manage to raise the pendulum. The delivered current from the motor controller card should probably be the next place to check. A current at about 2 A seemed to be sufficient, but since the torque of the pendulum depends on this current, it should be tuned to match the control system.

Later test runs with the Baldor card, showed no problems on moving the pendulum, as long as the current limit potmeter (see Section 4.3.2) is tuned correctly.

6.3 COM ports

The Wafer-9371A is equipped with two serial (COM) ports, where one is capable of receiving RS-232 signals, and the other could be configured to receive RS-232, RS-422 or RS-485 by modifying the settings on jumper JP4 on the SBC (Single-board computer), as shown in Table 3.3.

Some problems occured with these serial ports during this project. The first one to fail was the COM2 port. The jumper configuring the second serial port was checked. Jumpers, connecting pin 1-3 and 2-4 were installed. Due to Table 3.3 this is introducing some ambiguity. The 2-4 connection is indicating that the RS-244 is used, and the 1-3 connection set RS-232. It is worth noting that the RS-232 connection was actually working under these circumstances earlier in the project. The jumper closing pin 2-4 was removed in order to be sure to use the RS-232 signal.

The cable used to connect to the COM2 port is shown in Figure 6.5. All lines were tested by the "beeping" function on the multimeter, to check for some broken wires or connections, but it seemed to be fine.

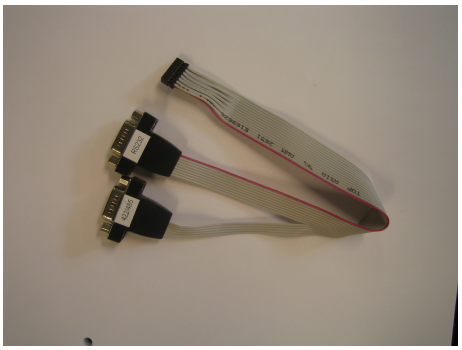

Figure 6.5: The serial cable provided with the Wafer-9371A, to be connected to the COM2 header.

Then the BIOS settings were evaluated, to be sure that no other devices was set up to use the specified address for the ports. The settings were COM1 on 3F8 and COM2 on 2F8, as expected, and no IrDA or ASK IR was enabled.

This had to be tested in another operating system, to be sure QNX was not playing a part in this error.

6.3.1 Linux installation

Linux was installed at the compact flash card by using a USB memory stick as a source. The stick (shown in Figure 6.6) was borrowed from another student, ready configured and set up by the procedure described in [Canonical Ltd.]. The Fujitsu hard drive was disconnected, in order to be sure not to owerwrite the QNX installation, when installing Linux on the CF.

Figure 6.6: The USB memory stick used as source storage medium when installing Ubuntu Linux on CyberBikePC.

When powering up the CyberBikePC, "DEL" was hit to enter BIOS setup, and the USB stick was entered as the first boot device. "Save and exit" from the BIOS setup menu, made the installation process begin shortly afterwards. The installation procedure is made as a guided tour through the various settings by Ubuntu. The only differences made from the default settings, was to uncheck the "Ubuntu desktop" box, to get a

minimal installation[3], and to not set up a swap disk. The latter choise was based on the fact that a normal compact flash card does not allow more than 10,000 to 1,000,000 write cycles [Wikipedia], and that the number of writing cycles to a Linux swap disk would probably exceed this limit fast and shorten the expected lifetime of the CF card.

The CyberBikePC was booted into Linux as described in Appendix C, and a root user and a regular user *"cyberbikerider"* was created, both with password "deluxe"[4] to keep things simple.

6.3.2 Testing of COM2

The serial ports were tested by simply reading from the serial devices at the terminal prompt, like shown in Printout 6.2. Here the GPS device was used as the signal source for two reasons. The first one is that it sends nice ascii-coded messages at a slow and easy-to-read rate. Second; it does not need to receive any messages to start delivering data, in contrast to – for instance – the Spark Fun IMU, which needs to receive a 0x07 signal to start transmitting measurement output. Printout 6.2 shows how the procedure was done on the COM1 port at first, to ensure that the GPS was delivering signal, and that a command for setting the correct baud rate was used.

```
$ su
<enter password>
# stty -F /dev/ttyS0 ospeed 4800 ispeed 4800
# cat /dev/ttyS0
$GPGGA,004151.029 ...

...
<ctrl+c to stop reading from the GPS>
#
```

Printout 6.1: Testing the COM1 port before testing COM2.

Then the GPS serial cable was connected to the second com port, using the RS-232 connection at the serial cable shown in Figure 6.5. The same procedure was applied to this serial device, as stated in Printout 6.1. No output was printed to the terminal. By this it was concluded that the second serial port was defect.

Comment

During this test, the source of the problem was not found, and the problem was not corrected in any way. The return of the test was a strong indication that the error was

[3] This was actually not done correctly the first time, and the installation process stopped unfinished due to memory shortage on the CF card.

[4] This password was chosen by reading the front brand tag on the bike, which says "Øglænd Perfekt deLuxe".

```
# stty -F /dev/ttyS1 ospeed 4800 ispeed 4800
# cat /dev/ttyS1
<ctrl+c to stop waiting>
#
```

Printout 6.2: Testing the COM2 port.

residing in the hardware in the Wafer-9371A. Some testing and probing had already been performed, which could be the reason for the damage (see Section 7.4). Another possibility was the two jumpers connected on J4, but the writer of this thesis has not found any statements in [IEI Technology Corp., 2006] that indicates that this would be harmful to the Wafer-9371A.

6.3.3 Testing of COM1

Very late in this project, the first serial port also started to behave erroneously. The MTi driver was not able to get through its initialization process, and ended with the message "No device connected". But closer inspection of the output from the driver showed that it actually did open `/dev/ser1` without problems. It was not until the driver tried to send a message to put the device in config mode, the error was returned. A quick read on the serial device, using the command `cat /dev/ser1`, flushed loads of output to the screen, indicating that the device actually was transmitting something on the COM1 port.

Then the CyberBikePC was booted into DOS as described in Appendix D, with a terminal program `tplus1_4` from [Lightsey] loaded into the USB-stick. The terminal was tested with the GPS module connected to the serial port. Setting the baud rate to 4800 made the output from the GPS visible on the screen. Then the IMU from Spark Fun was tested. The baud rate was set to 57600, as done in Section 4.1.1.

Pressing the reset button on the IMU sends some characters to over the serial connection, and this was also the case this time. But trying to put the device in measurement state by sending 0x07 to the device, did not work.

6.3.4 Comment

It is not done sufficient testing to ensure that the COM port is corrupted. But there are strong indices that the transmit line on the port is damaged, since the GPS is acting as it should, but as soon as data is to be sent back to a device, it does not work.

6.4 GPS driver testing

While testing the GPS driver some abnormal results occured. The following debug process gained some knowledge about how to get the resource managers to work better,

and is therefore described here.

When `devc-gps` was running, the content in its registered device names were garbled, as could be seen in Printout 6.3.

```
# cd /dev/gps/out/
# cat altitude
X0000
X0000
X0000
X0000
X0000
X0000
X0000
X0000
X0000
X0000
X0000
X0000
X0000

# cat utctime/hour
ooooooooooooooooooooooooooooooooooooooooooooooooo
#
```

Printout 6.3: Reading the GPS device files.

The GPS was sending its data, and starting the driver in verbose mode showed that it was parsing the messages, and that it tried to write to its device files. The values can be seen in Printout 6.3.

After some debugging, it was discovered that if a fixed long value were put in the buffer to be written to the device, some of it was shown in the file, but parts of it was garbled.

This indicated that something else was writing to the memory location where the value was stored, before it was read out from the device file.

Thanks to Øyvind Bjørnson-Langen, which once had a similar problem, the error was found. The buffer that the *gps_read()* function was writing the output data to, was defined as a local variable to that function. At the end of the function, an IOV (see Section 2.2.1) was set up to send the data back to the requesting client. This implies that the address of the buffer is handed over to the resource manager utility to be read back to the device file. Then the *gps_read()* returns, and its occupied memory space is free to be used by other functions. When finally the data is transmitted to the receiving client, some other function had been writing to to the memory where the return buffer was stored, and the data was corrupted.

Furthermore; to make the buffer a *global* variable instead of an autovariable, solved the problem. By closer inspection of the other resource managers developed for the CyberBike so far, this error seemed to be done in *all* of them. But obviously, it have not made itself visible in all cases.

6.5 DMM-32-AT issues

When the porpulsion motor was connected to the Baldor TFM 060-06-01-3 (see Section 4.3.2), some calibration of it was needed. Then it was discovered that the DMM-32-AT was giving erroneous output from its D/A converter. This section covers a description of the debugging process of this card.

When performing a write operation on the DMM-32-AT-resource manager `devc-dmm32at`, like shown in Printout 4.10, the output on J3-pin 38 was not as expected. The pin were measured by an oscilloscope (see Section 6.1.2).

Possible errors were:

- Hardware error in the DMM-32-AT, as a consequence of possible connection mistakes, causing too high currents or voltages on the output ports.
- Hardware error in the Wafer-9371A, giving the I/O card wrong instructions via the PC/104 connector.
- Software error in the way QNX is communicating with the I/O card.
- Software error in the resource manager `devc-dmm32at`.

6.5.1 The DOS Diagnostic Test Utility from Diamond Systems

By using the DOS Diagnostic Test utility included in `Util-dscud5.91.zip`, downloaded from [Diamond Systems Corporation, 2007] the possibility for errors in the `devc-dmm32at` could be tested (see Appendix D).

Running the menu element; "9. D/A 'QUICK' Diagnostic Test", gives the result shown in Printout 6.4. Both for A/D and D/A the tests passed, and the range was correctly set to $\pm 5V$.

```
D/A is detected as -5.00V to 5.00V. If not, then board has error or
is not calibrated correctly. (...)

*================== D/A Diagnostic Test ==================*
D/A Output   Expected A/D   Sampled A/D   Difference   Pass/Fail
         0         -16384        -16388            4        PASS
      1000          -8385         -8382            3        PASS
      2000           -386          -381            5        PASS
      3000           7612          7618            6        PASS
      4000          15611         15622           11        PASS
```

Printout 6.4: DOS Diagnostic Test, using the "D/A 'Quick' Diagnostic test output", while no probes are attached at the D/A channel 0.

Then the positive probe of the oscilloscope was connected to J3 PIN38 to measure Vout 0 (PIN40, Agnd, was already connected to the oscilloscope's negative terminal).

Running the same feature again gives a quite different output than earlier, shown in Printout 6.5

```
Da is detected as -5.00V to 0.25V (...)

*================== D/A Diagnostic Test ==================*

D/A Output   Expected A/D   Sampled A/D   Difference   Pass/Fail
         0         -16387        -16385            2        PASS
      1000         -12188         -8385         3803        FAIL
      2000          -7990          -383         7607        FAIL
      3000          -3791           816         4607        FAIL
      4000            406           820          414        FAIL
```

Printout 6.5: DOS Diagnostic Test, using the "D/A 'Quick' Diagnostic test output", while a probe are attached at the D/A channel 0.

The same behaviour can be observed when using the menu elements "4. D/A Autocal Test" and "10. D/A User Select Test": It looks correct when the positive oscilloscope probe is unconnected to Vout 0, and gets wrong when it is connected. Setting the value to 4095 by "10. D/A User Select Test" while probe is unconnected gives a correct behaviour on screen (range $\pm 5V$ and it says the output voltage is set to 4.995V). But the voltage is measured to be about 0.25V. Giving code 0 gives -5V output in any case.

Channel 3 and 1 works fine independent of the oscilloscope being connected, and channel 2 doesn't work at all. But it is only channel one who changes the range printed to the screen dependent on if it is connected or not.

In Section 3.1.4 it was stated that the BASE +15 is controlling an EEPROM Access Key Register. In [Diamond Systems Corporation, 2003, p. 25] it is stated that:

> *"The user must write the value 0xA5 (binary 10100101) to this register each time after setting the PAGE bit in order to get access to the EEPROM. This helps prevent accidental corruption of the EEPROM contents."*

Assuming the PAGE bits mentioned is the last two bits in the MISCELLANEOUS CONTROL REGISTER (BASE + 8); P1 and P0, this might be an error in the in the *init_ dmm32at()* function, in the original `devc-dmm32at` driver code (see Code sample 6.1).

Code sample 6.1 The contents of *init_ dmm32at()* in the DMM-32-AT driver; `devc-dmm32at`.

```
out8(BASE_ADDR+MISC_CTL_REG,0x23);     // reset and calibration
out8(BASE_ADDR+MISC_CTL_REG,0x00);     // (un-reset)
out8(BASE_ADDR+OP_CTL_REG,0x00);       // operation ctrl reg
out8(BASE_ADDR+AN_CONF_REG,0x39);      // -5-5V bipolar
```

In the first code line the value 0x23 (binary 100011) is set to this register. The last two bits is controlling the paging of the calibration device to the last 4 registers in Table 3.8. It is set to zero straight afterwards in the second line, but an error might have happened as a consequence of not writing the specified value 0xA5 to the EEPROM Access Key Register.

Another possible, but probably insignificant error in the original `devc-dmm32at` driver, is that the FIFO memory never becomes reset. This should be done each time before an A/D operation is started, in order to be sure to get the current data. In the case of the current driver the FIFO is not enabled, but actually it is still used in an one-to-one correspondence between sampling and reading, where the FIFO contents never exceed one sample.

6.6 Testing the control algorithm

The control system made by Bjermeland [2006] was tested – within a limited amount – with the hardware. Matlab S-function for reading data from the `devc-velo` resource manager was implemented, and a first attempt to connect the data from the control algorithm was made.

Block diagram for the main blocks in the Cyberbike simulink model is shown in Figure 6.7. The main blocks is – except "The CyberBike" – just a rearrangement of the system made by Bjermeland [2006]. The CyberBike block contains the S-function block performing the driver communication, and signal selection, scaling and integrating.

For a tutorial in how to compile and use the S-functions, refer to appendices in [Fossum, 2006]. The main code in the bike_io S-function is found in the function *bike_io_Outputs_wrapper()* in `bike_io_wrapper.c`, and is shown in Code sample 6.2.

The device files are opened and closed in two functions, called upon initialization and termination respectively. These functions are shown in Code sample 6.3

Various results occured during the test runs of the code, depending on how many devices the system was specified to sample from, and and how much of the model of Bjermeland [2006] that was included. The current model, appended on the CDROM, is terminated after just a few samples. There has not been done enough testing to point out where the problem resides. The first that came in mind, was that the sampling rate was too high (0.02s) compared to the calculation/computing task that had to be performed. The Wafer-9371A CPU is only running at 400 MHz, which is no problem to exhaust, running a complex matlab code with observer and a bike model.

Further testing on this was discontinued by the occurrence of the last COM port problems, discussed in Section 6.3.2.

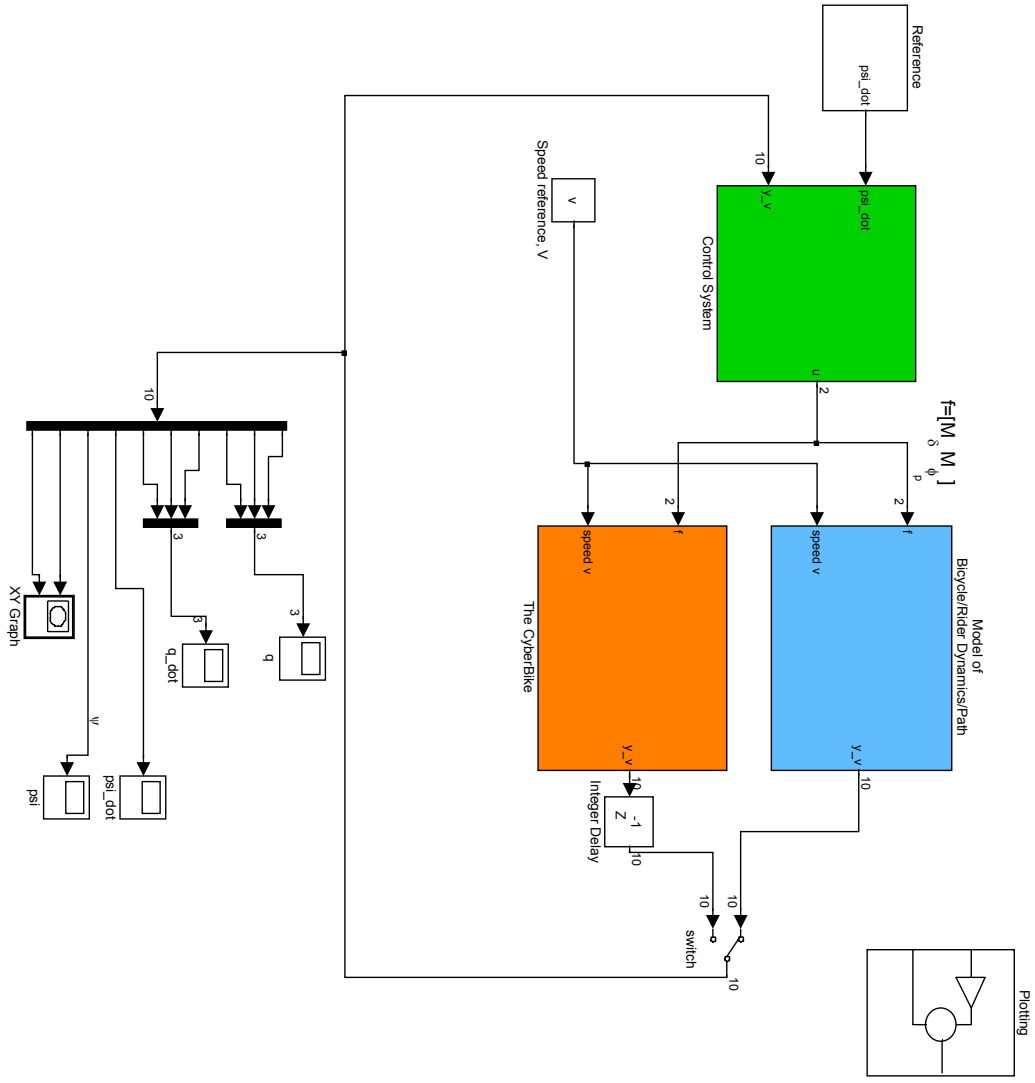

Figure 6.7: The main blocks in the CyberBike Simulink model.

Code sample 6.2 The main code performing the I/O operations to the actual bike in `bike_io_wrapper.c`.

```
//————————————————————————————————//
//       Reading from the CyberBike:          //
//————————————————————————————————//
for (i = 3; i < NumVeloDevices; i++ ) {
    ret = devctl( fd[i], VELO_DEVCTL_GETVAL, &meas, sizeof(meas), NULL );
    if ( ret == -1 )
        fprintf(stderr, "%s:\tERROR when performing devctl(getval) \
                         %on %s:\t%s\n", NAME, velo_devnames[i], strerror(ret) );
    else
        *(devs[i]) = (real_T) meas;
}

//————————————————————————————————//
//       Setting the output:                  //
//————————————————————————————————//

for ( i = 0; i < 2; i++ ) {
    output = *(devs[i]);
    ret = devctl(fd[0], VELO_DEVCTL_SETVAL, &output, sizeof(output), NULL);
    if ( ret == -1 )
        fprintf(stderr, "%s:\tERROR when performing \
          devctl on %s:\t%s\n", NAME, velo_devnames[i], strerror(ret) );
}
```

Code sample 6.3 The *bike_start()* and *bike_stop()* functions in `bike_io_wrapper.c`.

```
int bike_start() {
  int i;
  int openflag = 0;
  printf("Running bike_start() procedure.\n");

  // the first 3 devices are motors, and needs to be
  for (i = 0; i < NumVeloDevices; i++) {
    openflag = ( i < 3 ? O_RDWR : O_RDONLY );     // The first 3 devices are motors,
                                                  // and will be written to
    fd[i] = open( velo_devnames[i], openflag);
    if (fd[i] == -1) {
      perror("open() returned with error:");
      return fd[i];
    }
  }
  return 0;
}

int bike_stop() {
  int i;
  int retval = 0;
  double output = 0;
  printf("Running bike_stop() procedure.\n");
  for (i = 0; i < NumVeloDevices; i++ ) {
    retval |= devctl(fd[0], VELO_DEVCTL_SETVAL, &output, sizeof(output), NULL);
    if ( retval == -1 )
    fprintf(stderr, "%s:\tERROR when performing devctl on %%s:\t%s\n",
                              NAME, velo_devnames[i], strerror(retval) );
    if(close(fd[i]) == -1)
    retval |= EXIT_FAILURE;
  }
  return retval;
}
```

Chapter 7

Discussion

An overview of the CyberBike's hardware situation at the end of the work on this thesis is illustrated in Figure 7.1. It is put here to be comparable to Figure 3.1, to see the achieved hardware extensions during this work. Switches and buttons are not shown.

7.1 Choice of storage medium

The reasons for choosing the hard drive instead of the flash card have been described in detail in Section 4.2.1. Generally it could be stated that the choice was made to get a quick solution to the OS-image problems. The disk is mounted at the suitcase's bottom plate, beneath the motor controller card.

One thing that have been left to possibly be discovered during future testing, is if the hard drive could put up with the disturbances imposed to it. The suitcase and its content would probably be exposed to shaking and mechanical vibration from the pendulum and rapid changes in steering angle, wheels running at an uneven surface could cause vertical perturbations, and even worse; if the bike collides or tips over, many kinds of mechanical shocks could occur.

A better mounting solution might partly solve the problem. An example is shown in Figure 7.2, where elastic rubber bands (or another suitable material) is used to damp the disk housing accelerations, relative to the bikes movements.

The reason for not mounting it this way, was partly because there is little space left in the suitcase. But, unfortunately, the most crucial reason was that it was not thoroughly considered when the disk was mounted. This could lead to future problems, but was not yielding any problems during the work on this thesis.

7.2 IMU Choice

When the first IMU (from Spark Fun) was ready and connected to the Simulink controller via S-functions, it was realized that a lot of tuning and calculations of the IMU-data had

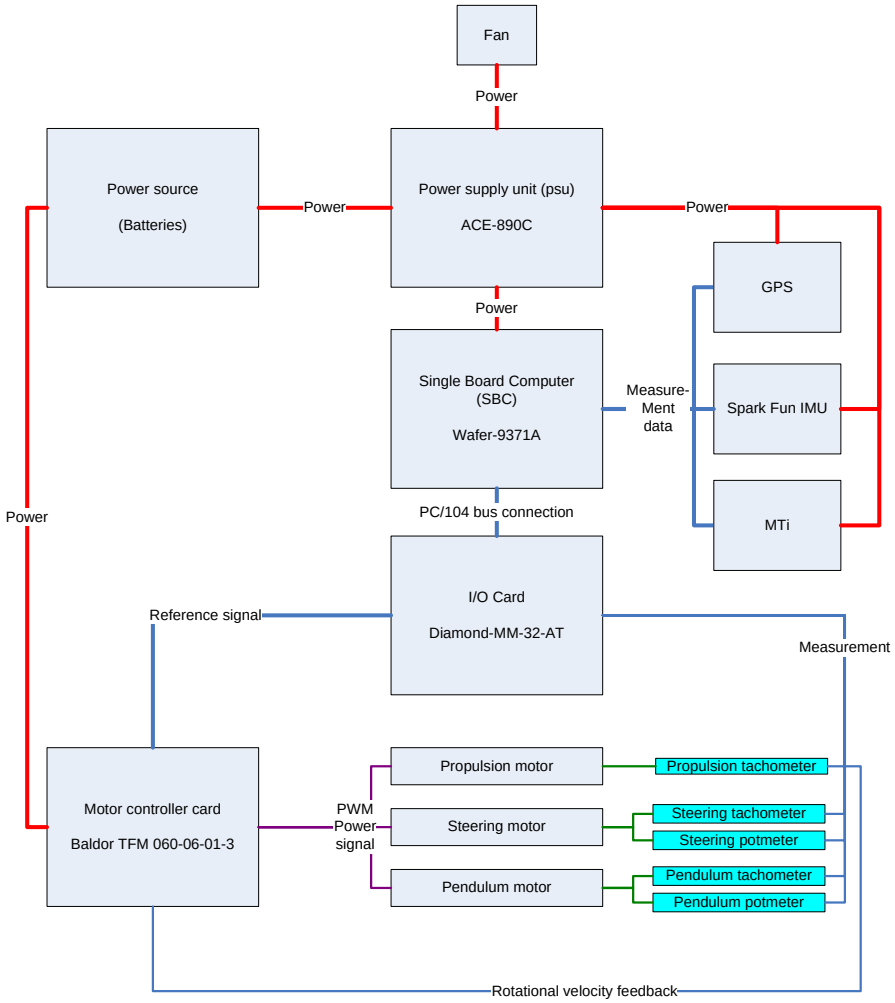

Figure 7.1: Overview of the CyberBike's various hardware devices by the end of this work.

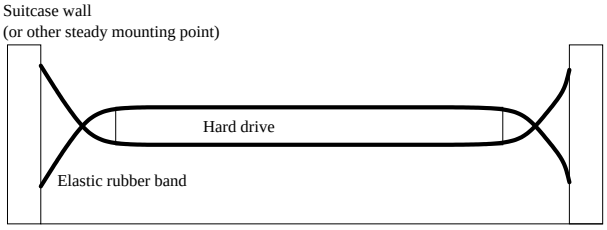

Figure 7.2: Suggestion for better physical mounting of the disk drive.

to be done in order to make them useful. The MTi contains its own digital signal processor, giving the user the possibilities to choose between different representations of the data. If Euler angles is chosen, as is the case for the CyberBike, the device is sending calibrated ready-to-use *"roll-pitch-yaw angles"* to the host. It also supports different user-specified baud rates and sensor-update frequencies, making it possible to optimize it better for the specified task.

Consequently, there is reason to believe that the MTi would give a better result, with less effort put into it, than the Spark Fun IMU. Hence; it seems like a wise decision to start using the MTi instead[1] of the original IMU.

Perhaps the MTi should have been included at an earlier stage, in order to get closer to the goal of the project. It should not be necessary to connect two IMU to the bike at the same time, and if the MTi is present, the original IMU is superfluous. A voting mechanism could be implemented if both are to be used, but for a non-critical application like the CyberBike, this is not very interesting, generating an even more complex system, with enhanced error possibilities.

7.3 GPS

When the GPS module was ordered, it was not clear whether it should be used at all, but it seemed like a practical and easy solution to order an extra similar module, when Eriksen [2007] was sending his orders anyway. He also made the layout of the PCB to connect it to a serial port, which made creating the hardware easier. Using his software for this project was also considered, but when connecting the GPS to the CyberBikePC was about to be done, using the resource manager for the IMU as a framework to get a QNX compatible driver seemed easier than to modify and integrate Eriksen's LabView solution.

The GPS functionality implementation process, including making the PCB, solder the components, making the communicating software and mounting the unit to the suitcase went reasonably well, without too much time spent on debugging and coding. Still;

[1] Actually; the Spark Fun IMU is still mounted on the CyberBike, in case the MTi gets unavailable. The MTi is at this point in time as mentioned in Section 4.1.3, shared with another project.

reading and understanding the module's manual, writing the message parsing code, and etching and soldering the PCB takes some time.

At the end, the information from the GPS was not used in the control algorithm as planned. Maybe it was unwise to spend time at all on this task, but the time spent on the GPS-development at this stage, is hopefully saved in the subsequent.

7.4 Debugging hardware

As noted earlier, for instance in Section 4.5, Section 6.5 and Section 6.3, some hardware related errors have occured during the project. The downside of such errors span much wider than just the "wasted" time on debugging. In situations, like the one described in Section 4.5, the complete overview of what the various connections in the system actually does was lost. This leaves the system in a very dangerous state. Voltages and currents above the absolute maximum ratings for the hardware may destroy iCs and other circuit components.

A human weakness is that they make mistakes. Long series of probing with multimeter may sooner or later result in short circuiting of pins lying close to one another, or probings with the multimeter set up incorrectly.

Therefore, the connections in the system should be kept as easy to understand and as well arranged as possible. If an earlier realization of how much additional connections and probing that had to be done, a replacement of the connection system in the CyberBike's suitcase might have been a good idea. An example is to replace the long pins from the Baldor TFM 060-06-01-3 with a connector for soldering on a PCB. The circuit board would hold all the wires and route them correctly to connectors, which in turn could be connected by short dedicated ribbon cables to DMM-32-AT and the terminal block on the suitcase's outside.

The layout of the PCB could have been simulated in software, to point out and remove a lot of the errors before connecting it to the physical system. Using a *Programmable Logic Device* (PLD) such as a FPGA's or GAL could also be done, to ease tasks like the pendulum limit switches, or simply to "wire" the connections between two points in software, before burning the circuit to the PLD.

7.5 Reflection

Defects on the I/O-card (DMM-32-AT), used for data acquisition (A/D converting) and to output motor reference voltages (D/A converting), were pointed out during the work on this thesis. These errors entry-time and -cause are unknown, but what *is* known is that it resulted in a considerable retardation of the project's progression. At the end, the errors were partly bypassed by using channels that did work.

It turned out that the amount of errors inherent in the system, were larger than expected at the starting point of the thesis' work. This, in conjunction with the I/O-card

and COM port problems, and the late change of IMU-strategy, made too big impediments to reach the goal of a real outdoor CyberBike autonomous test run.

With the benefit of hindsight, some questions that should be answered here is; Were the choices made during this thesis reasonable? Would other choices have got us closer to the goal?

As already pointed out, the tasks involving GPS was performed somewhat too early, considering that it ended up unused by the control algorithm by the end point of this thesis. But if the CyberBike-project is seen as a whole, the introduction of the GPS is indeed useful, and it was believed that the time left would be sufficient to implement useful functionality around it. Also; introducing the MTi at an earlier stage would have saved some development time on the original IMU. This is maybe the most grave compliant to the completion of the work in this thesis. But still it could be argued that the IMU had to be tested and used to explore its qualities and limitations.

Time is obviously generally involved in most of the issues discussed in this chapter. The thesis' writer hereby take the opportunity to emphasize that debugging in instrumentation systems like the CyberBike's *do take a lot of time*. It is important to note that a lesson learned by the work in this thesis, is that very often *none* of the parts in the system could be fully trusted. Using several hours trying to find errors in newly self-written software – usually considered to be the most obvious place to start – more than once resulted in discovering that the error was lying in a hardware unit; fully operative until recently and believed to be thoroughly tested.

Chapter 8

Further Work

This chapter describes what has to be done next, and what could be interesting to get implemented. An attempt to set the various tasks in a prioritized order is suggested here.

The first task to be assigned to the projects subsequent student, should be to get the defect hardware to work properly. This includes the serial ports on the Wafer-9371A(see the following section), and the output channels on DMM-32-AT. The decision on how to solve it has to be made quite early in the work on a new thesis. If a solution involving recondition of existing hardware is to be carried out, make sure that the hardware will be returned within reasonable time.

This said; it is believed that a fifth year student at ITK, effectively and rationally putting an effort into the CyberBike-project, during the spring term project, would be able to execute the bikes first autonomous test run.

8.1 Correction of the serial connection problem

The problems in Section 6.3 need to be solved in some way or another. Maybe the Wafer-9371A vendor IEI Technology Corp. is able to check and repair the board for a reasonable price, but this has to be checked before mailing it all the way to Taiwan.

As the situation is right now, there are three devices to be connected to the CyberBikePC. The two serial ports provided on the board are not sufficient to connect both IMUs and the GPS at the same time. In that case some kind of serial port extension should be purchased. An example of such a board is the CA-104 from MOXA [AS], which would fit in the PC/104 stack in the CyberBike's suitcase. A USB-to-serial device might not be a good option, owing to the fact that such devices have proved difficult to cooperate in QNX (see Section 4.1.3).

However; this might not be of interest if the COM ports on the SBC gets repaired, since one of the IMUs should be able to do the job. The cost of the Wafer-9371A reconditioning must be weighted against the price on a new serial port extension board.

8.2 Wireless ethernet connection

The CyberBike is supposed to move around without any wires attached. By now Matlab, Simulink and Real-Time workshop have been used to control the bike from a host machine. To be able to keep the system like this, installation of a wireless ethernet adapter on the bike seems like the most obvious solution. Some kind of remote control should be implemented anyway, and an ethernet connection could be used to cover most of the demands in one link.

An antenna to be used for radio commuication is installed on the bike, but is not being used recently. But to base a solution on this technology would lead to difficulties in allocating enough bandwith to perform tasks including Real-Time Workshop and the like, mentioned earlier.

The easiest solution is probably a device to be plugged into the RJ-45 connector on the outer side of the suitcase, sending the signal further through the air. Then searching for WiFi cards with QNX compatible device drivers would not be an issue.

8.3 Line of sight

The GPS is now mounted and is communicating with the CyberBikePC[1], but not used for anything particular. Bjermeland [2006] states that a *Line Of Sight* (LOS) algorithm would fit for the bike. Then a route could be given to the CyberBike by defining some waypoints, and the placement calculations – developed from the speed and heading angle measurements – would be used to control the bike to reach these points. The GPS would be necessary to correct and update the placement calculations along the way. For a further description of the LOS algorithm, refer to [Fossen, 2002].

8.4 Optimization of `devc-dmm32at`

The DMM-32-AT is not utilized the way it could. The card has got the capabilitiy of using a FIFO memory, scan sampling with a programmable time between samples between 20 and $5\mu s$, interrupt based sampling for controlled-rate sampling, and channel-range sampling. The CyberBike's control algorithm is supposed to use a constant sampling rate of all input-signals for every step, and this could have been performed a lot more efficient by using some of these features provided by the I/O card.

In [Diamond Systems Corporation, 2003, p. 31] an overview of different sampling methods are described, and for which types of application the different methods fit.

But before this is investigated further, the CPU utilization for the driver should be evaluated to decide if this is really needed. If the current sampling method makes the driver satisfy the real-time constraints (considerations on delay and jitter[2]) for the

[1] as soon as more serial interfaces are added to the system

[2] variation in delay

CyberBike, and the total CPU utilization on the CyberBikePC is sufficiently low when all the necessary drivers are running, it might not be worth the effort.

8.5 Videocamera

It would be interesting to see and record the trip from the CyberBike's point of view, when the bike is going for an autonomous ride. For this purpose a simple web camera would be sufficient. An analog camera with its own unit for transmitting data is assumed to be the simplest solution here, to ease installation and to avoid an overload of the WiFi connection and the CyberBikePC's CPU consumption. A webcamera with USB interface would require support from the QNX Neutrino system running on the CyberBikePC, and hence might also generate some additional compatibility problems. To get the best possible real-time properties on the system, the load on the CPU should be lower than 100%, and this is also a relevant argument for not letting the camera signal go through the CyberBikePC's CPU.

8.6 Adaptive IMU-signal converting

In Section 7.2 it was argued that the MTi is a far better solution than the original IMU from Spark Fun. But since the original IMU already is purchased and available, it could be interesting to put some effort into making it a better device, releasing one or the other to other projects. If an algorithm, similar to the one running at the MTi's DSP, could be derived, the IMU's utility value would increase.

From [Xsens, 2006] it could be read that the physical sensors inside the MTi are calibrated according to a physical model of the sensors' response to various physical quantities. Temperature is named as an example. The combo boards (the three cards with embedded accelerometers and gyros) on the Spark Fun IMU also provides temperature measurements, included in the 16 values transmitted from the IMU motherboard. Xsens [2006] states that the basic model used at the MTi is linear, and according to the relation given in Equation (8.1), where $\mathbf{K}_T$ is a unique gain matrix, given by the factory calibrations, $\mathbf{b}_T$ is the bias vector, $\mathbf{u}$ is the sampled voltages.

$$\mathbf{s} = \mathbf{K}_T^{-1}(\mathbf{u} - \mathbf{b}_T) \tag{8.1}$$

It is also noted that the model actually used by Xsens is much more complicated than the Equation (8.1), and that their model is continously being developed.

An alternative way to develop the model, or the parameters in the above equation, would be to use an adaptive algorithm, reading the data from the IMU and from the actual known accelerations and rotation speeds.

The suggestion for further work given in this section, is not yielding the CyberBike project any real progression as long as the MTi is available. It is mentioned here because

it might be an interesting task to be assigned as a separate project or thesis in the future, or if sharing the MTi between two or more projects gets too complicated.

8.7 Connection database

If serious reorganizing or adding of several connections and devices to the CyberBike's suitcase, it might be worth to consider using a database tool to keep the documentation of the connections up to date. In Appendix E some tables showing the connections are presented. These probably get out of date as soon as hardware related upgrades are done to the CyberBike. Parts of the information are presented in more than one table. This complicates the update process of the documentation, and makes it harder to keep an overview. However, if changes to the hardware has to be made, an update of the existing tables is probably preferable.

Onshus [2006, chap. 16] is describing how documentation of large control systems should be done, and mentions a few examples on computer assisted documentation software packages. Using these would probably be an overkill in this case, but maybe simpler alternatives exist to be used in the CyberBike's case.

Chapter 9

Conclusion

A project on an autonomous unmanned bicycle – the CyberBike – has been going on at the Department of Engineering Cybernetics (ITK) at NTNU for some years. The goal of the project is to make the bike ride on its own, by controlling the steerangle, the bike's velocity, and an inverted pendulum substituting the leaning rider. To achieve this, three motors are used to apply torque to the steering, the rear wheel, and the pendulum. But to control these, measurement devices and a calculating unit is necessary, as well as a controller.

The work on this thesis includes various tasks, mainly focusing on software and hardware development for the instrumentation parts of the control system. A part of the assignment, was to find out what had to be done. Measurements from two new devices have been made available to the controller; an *Inertial Measurement Unit* (IMU) and a *Global Positioning System* module (GPS). Some devices already included or partly attached to the system needed further development to work properly with the rest of the system. This introduced code and hardware maintenace tasks as some of the biggest and most demanding parts of the work to be done. To be more concise; the driver for the I/O card (DMM-32-AT) needed a thorough investigation and rewriting, the propulsion motor connection and setup had to be completed, the IMU initially used in the CyberBike project (from Spark Fun Electronics) needed a UART-to-RS232 signal transceiver circuit to enable measurement transmission. The same IMU also needed some code maintenance to work properly, and the pendulum motor needed functionality implemented to its already mounted limit switches.

Finally some pure hardware related tasks were carried out. These included installation of a fan for cooling the electronic devices in the control system, batteries to power up the system without using cables, and an emergency stop switch.

Some attempts to use the measurements from the various devices together with a previously developed CyberBike model and controller in Simulink, using S-functions, were performed. But to get any specific test results from this, not sufficient time was left for the exercise.

When the first IMU (from Spark Fun) was installed and tested, it was realized that

a large effort on converting and calibrating the data received from the unit had to be done, to be able to use the measurements to controlling the bike. It was then decided to try the other and more advanced IMU (the MTi) instead. This unit contains its own digital signal processor, converting the measurements to ready-to-use variables directly from the device, hence better results were expected by using this unit.

By this, it is believed that the CyberBike-project is far closer to its finite goal, yielding a better foundation for further development and progression on the project.

References

ACT Batteries. [online]. URL http://www.actbatteries.co.uk/. [Accessed 30 May 2007.

Elektronix AS. Ca-104. [online]. URL http://www.elektronix.no/Produkter/Kommunikasjon_RS_Ethernet/PC-_basert_serie_kommunikasjon/Multi_port_serie_kort/Moxa_PC_104_-_forembeddedPC_systemer_/1167. [Accessed 4 Jun 2007].

Baldor ASR. *Pulse Width Modulated Transistor Servodriver TFM Instruction Manual.* Baldor ASR GmbH, Dieselstraße 22, D-8011 Kirchheim-München, West Germany, 1988. Version 1:11/07/88.

Lasse Bjermeland. Modeling, simulation and control system for an autonomous bicycle. Master's thesis, Norwegian University of Science and Technology (NTNU), Trondheim, June 2006.

Edgar Bjørntvedt. Instrumentering for autonomt ubemannet fly. Master's thesis, Norges teknisk-naturvitenskapelige universitet (NTNU), Trondheim, June 2007. Working title only. The master thesis was unpublished when this thesis was written.

Canonical Ltd. LiveUsbPendrivePersistent. [online]. URL https://wiki.ubuntu.com/LiveUsbPendrivePersistent. [Accessed 3 Jun 2007].

Diamond Systems Corporation. *Diamond-MM-32-AT 16-Bit Analog I/O PC/104 Module with Autocalibration, User Manual, V2.64.* 8430-D Central Ave., Newark, CA 94560, 2003. URL http://www.diamondsystems.com.

Diamond Systems Corporation. Online support: manuals, drivers, technical community. [online], May 2007. URL http://www.diamondsystems.com/support/. [Accessed 23 May 2007].

DtC-Lenze as. [online], May 2007. URL http://www.dtc.no. [Accessed 25 May 2007].

EAO. *Condensed Catalogue, Experts in Human Machine Interfaces.* EAO AG, Tannwaldstrasse 88, 4601 Olten, Switzerland, 2007. URL http://www.eao.com/global/en/products/condensedcatalogue/byseries/UK_English/EAO_SERIES_44_SWITCHES_ENGLISH.pdf. [Downloaded from vendor web site 24 May 2007].

Olav Egeland and Jan Tommy Gravdahl. *Modeling and Simulation for Automatic Control.* Marine Cybernetics AS, P.O. Box 4607, NO-7451 Trondheim, Norway, second edition, Jun 2002. URL `http://www.marinecybernetics.com`.

ELFA. [online], 2007. URL `http://www.elfa.se/no/`. [Accessed 9 Mar 2007].

Mikael K. Eriksen. AUAV Ground Station «correct this reference !!!». Master's thesis, Norwegian University of Science and Technology (NTNU), Trondheim, June 2007. Working title only. The master thesis was unpublished when this thesis was written.

Thor I. Fossen. *Marine Control Systems: Guidance, Navigation and Control of Ships, Rigs, and Underwater Vehicles.* Marine Cybernetics AS, P.O. Box 4607, NO-7451 Trondheim, Norway, 1 edition, Dec 2002. URL `http://www.marinecybernetics.com`.

John A. Fossum. Instrumentering og datasystemer for autonom kybernetisk sykkel. Master's thesis, Norges teknisk-naturvitenskapelige universitet (NTNU), Trondheim, Dec 2006.

Fujitsu. *MHJ2181AT, MHK2120AT, MHK2090AT, MHK2060AT DISK DRIVES PRODUCT MANUAL.* Fujitsu Limited, Fujitsu Learning Media Limited, 22-7 Minami-Ooi 6-Chome, Shinagawa-Ku, Tokyo 140-0013, JAPAN, 1999. Manual code C141-E088-03EN.

GlobalSat Technology Corporation. *PRODUCT USER MANUAL, GPS RECEIVER ENGINE BOARD, EM-411.* 16, No.186,Chien 1 Road, 235Chung Ho City,Taipei Hsien, Taiwan ,R.O.C. URL `www.globalsat.com.tw`.

Stephen Hewitt. [online], 2002. URL `http://www.angelfire.com/falcon/speedload/Enabler.htm`. [Accessed 29 May 2007].

IEI Technology Corp. *Wafer-9371A 3.5" Profile Wafer Embedded Board, User Manual, Rev 1.1*, Jan 2006.

IEI Technology Corp. 86W DC24V Input Open Frame AT Power Supply. [online], 2005. URL `http://www.ieiworld.com/en/Product_IPC.asp?model=ACE-890C`. [Accessed 30 May 2007].

Rob Krten. *Getting started with QNX Neutrino 2.00 : a guide for realtime programmers.* PARSE Software Devices, 278 Equestrian Drive Kanata, Ontario, K2M 1C5, CANADA, 2nd edition, Aug 2001. ISBN 0-9682501-1-4. URL `http://www.parse.com`.

Lenze. *Lenze Operating Instructions.* Lenze GmbH & Co KG, Small drives, Postfach 10 13 52, D-31763 Hameln, 1.0 edition, Jun 2002. URL `http://www.dtc.no/filer/Manualer%20Lenze/Manual%20Sm%E5motorer%20AC-DC-PM%2Bgear%20GB.pdf`.

Scott Lightsey. tplus1_4.zip. [online]. URL `http://www.simtel.net/product.php[id]13585[cid]47[SiteID]simtel.net`. [Accessed 3 Jun 2007].

Hans Olav Loftum. Styresystem for kybernetisk sykkel - instrumentering for styring av en tohjuls herresykkel. Master's thesis, Norges teknisk-naturvitenskapelige universitet (NTNU), Trondheim, June 2006.

Joachim Marder. [online], Jul 2001. URL `http://www.jam-software.com/software.html`. [Accessed 27 May 2007].

Maxim/Dallas Semiconductors. MAXIM +5V-Powered, Multichannel RS232 Drivers/Receivers. [Online], 2006. URL `http://datasheets.maxim-ic.com/en/ds/MAX220-MAX249.pdf`. [Accessed 06 Mar 2007].

Tor Onshus. *Instrumenteringssystemer*. Institutt for teknisk kybernetikk, NTNU - Norges teknisk-naturvitenskapelige universitet, Trondheim, 4 edition, Januar 2006. URL `http://itk.ntnu.no/ansatte/Onshus\_Tor/`.

OpenQNX. OpenQNX: The QNX Community Portal Site. [online]. URL `http://www.openqnx.com`. [Accessed 3 Jun 2007.

QSSL. *10 Steps to your first QNX program, Quickstart guide*. QNX Software Systems, 127 Terence Matthews Crescent, Ottawa, Ontario, Canada, K2M 1W8, second edition, Sept 2005.

QSSL. Usb software development kit. [online]. URL `http://www.qnx.com/developers/docs/6.3.0SP3/ddk_en/usb/copyright.html`. [Accessed 4 Jun 2004].

QSSL. Qnx developer support - qnx documentation library. [online], 2007c. URL `http://www.qnx.com/developers/docs/momentics621\_docs/neutrino/lib\_ref/`. [Accessed 23 Apr 2007].

Spark Fun. Your source for prototyping supplies. [online], 2005. URL `http://www.sparkfun.com`. [Accessed 16 Feb 2007].

the free Encyclopedia Wikipedia. Compactflash. [online]. URL `http://en.wikipedia.org/wiki/Compact_flash`. [Accessed 3 Jun 2007].

Xsens. *MTi and MTx Low-Level Communication Documentation, Revision E.* Xsens Technologies B.V., Capitool 50, P.O. Box 545, 7500 AM Enschede, The Netherlands, March 2 2005. URL `http://www.xsens.com`.

Xsens. *MTi and MTx User Manual and Technical Documentation.* Xsens Technologies B.V., Capitool 50, P.O. Box 545, 7500 AM Enschede, The Netherlands, revision g edition, March 2 2006. URL `http://www.xsens.com`.

Litterature

Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages*. International Computer Science Series. Pearson Education Limited, Edinburgh Gate, Harlow, Essex CM20 2JE, England, 2001. URL `http://www.pearsoned.co.uk`.

Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Software Series. Prentice Hall P T R, Prentice-Hall, Inc., Upper Saddle River, NJ 07458, second edition, 1988.

William Stallings. *Operating Systems, Internals and Design Principles*. Pearson Prentice Hall - Pearson Education, Inc., Upper Saddle River, NJ 07458, 5 edition, 2005.

Appendix A

Contents on CDROM

The contents on the appended CDROM is:

Code contains the drivers and various testprograms for the bike. Available drivers are:

- devc-dmm32at
- devc-dmm32at-old
- devc-gps
- devc-imu
- devc-mt
- devc-velo

Connection_diagrams contains an Excel spreadsheet with the source of the various connection diagrams, presented in Appendix E, and generated LaTeX files.

Eagle-PCB-layout contains the Eagle-files for both signal transceiving PCBs made in this project (GPS and IMU).

Litterature includes electronic versions (pdf) of some of the articles, manuals and datasheets relevant for the CyberBike.

matlab contains files to make Real-Time Workshop cooperate with QNX.

Report contains this report, with figure sources, LaTeX-files, and final pdf.

Scripts contains some small scripts made to do often performed operations in QNX. Particularly the `nfs_script` have been useful.

Simulink contains some simulink models, with S-functions to communicate with the CyberBike's drivers.

Software contains software used, such as the various programs and libraries provided by the DMM-32-AT's vendor, programs to make a DOS bootable USB-stick, Terminals, NFS server program, etc.

Appendix B

How to start the CyberBike system

This chapter is intended to give a flying start of how to start the system.

1. Push the emergency button, and make it stay in activated position (no torque on motors)
2. Put the power switch in off-condition (a zero is visible at the top of the switch).
3. Connect a power source to the rightmost circular 4-pins connector at the suicase (marked 24V), either from the batteries, or from an external power supply (e.g. as showed in Section 6.1.3).
4. Make sure the MTi is inside it's housing right behind the front wheel.
5. Plug a screen cable (from an available screen) to the VGA connector on the Wafer-9371A (SBC).
6. Connect a keyboard into the PS2 connector at the Wafer-9371A.
7. Plug an ethernet cable into the RJ-45 slot on the outer side of the suitcase, and into an available network connection slot.
8. Connect a mouse (if wanted) into the USB connector at the outside of the suitcase. Make sure the USB cables connectors are connected to the 9-pins contact at the Wafer-board inside the suitcase.
9. Make also sure that the hard drive are connected to the EIDE[1] connector on the board. The hard drive is placed under the motor controller card (Baldor TFM 060-06-01-3).
10. Connect the desired serial device to the available com port (a D-SUB 9 connector) directly on the wafer board. In the present moment of writing, only this serial

[1] Enhanced IDE

device is working. The three available devices that could be plugged in are the MTi, the IMU from Spark Fun, and the GPS. If the MTi is chosen, make sure it is powered by the three-pins contact, from the ACE-890C. This connector is made such that it has ground on pin 1 and 3, and Vcc on the pin in the middle. This is to avoid destroying the device if plugging it in upside down.

11. Push the power button. If an external power source is used, it should be set to 24V. Make sure it could deliver enough current. Rememember that the motors are capable of using $11,8A + 2.9A + 4A = 18.7A$ at rated speed. The fuses on the batteries are chosen to be 15A, which should be more than sufficient for the load applied to the motors in this case.

12. If the suicase is closed (be careful if cables are hanging out) make sure the fan is running. The fan is a bit noisy, and therefore a switch, placed at the foremost left corner in the suitcase, is made such that the fan stops when the suitcase is open.

13. The QNX Neutrino login screen should appear on the screen. Press the "Superuser" icon, or type "root". No password is needed.

14. If another PC with the QNX Momentcs IDE are to be used for development, go through the QNX Quickstart guide [QSSL, 2005], to set up the host system (the target system; CyberBikePC, should not be altered).

15. The emergency button should now be deactivated by turning it counterclockwise, if some output on the motors are desired.

16. Start the drivers on the target machine by typing:

```
# devc-dmm32at
# devc-imu
# devc-mt
# devc-gps
# devc-velo
```

    in a terminal window. The `devc-velo` has to be started last. Device files should be accessible from the `/dev/` directory when the drivers are successfully started.

17. An NFS-server have been used to share files between the development host and the CyberBikePC. If this is not set up, an NFS-server is put on the CDROM appended with this thesis. Setup instructions could be found in [Fossum, 2006, Vedl. 5].

18. Some scripts to make working with the CyberBikePC easier is put on the CDROM (see Appendix A). For instance, the `nfs_script`, which takes the last octet in the IP-address as an argument (assuming the first three octets stays the same inside NTNU[2] for a while).

[2]Norwegian University of Science and Technology

19. To start developing from Matlab and Real-Time Workshop, see appendices in [Fossum, 2006] for quick introduction tutorials.

20. For developing in QNX, see Section 2.2, [Krten, 2001] and [QSSL, 2007c].

Appendix C

How to boot CyberBikePC in Linux

When testing the COM ports at the Wafer-9371A, another operating system was desired, to make sure that QNX was not the one to blame for the non-working COM2 port. An Ubuntu Linux-installation is now kept on the CF card (see Figure 3.5).

To boot into Linux, do the following:

- Power down the PC
- Remove the hard drive ribbon cable from the EIDE slot on the Wafer-9371A.
- Insert the compact flash card into it's slot inder the Wafer-9371A. Slim fingers are advantageous to get it in place.
- Power up the PC.
- Hit "DEL" to enter BIOS[1] setup.
- Choose "Auto-Detect Hard Disks". In the menu showing up, the CF card should be visible as secondary master.
- In the "Advanced CMOS Setup", make sure the PC is set up to boot from IDE-0, as it's first specified boot device.
- Save and exit BIOS setup.
- The CyberBikePC should now boot up in Linux. Login as `cyberbikerider` with password `deluxe` (same as the bicycle-brand). To get root privileges, run the command through `sudo`, or just type `su` to switch to super user. The super user password is the same as the regular user, to keep it simple, and because it normally would be the same person.
- To get back into QNX, type `halt` as superuser or through `sudo`, and shut the power off when told to.

[1] Basic Input/Output System

- Remove the CF card.
- Insert the gray hard drive ribbon cable into the 44 pins EIDE connector on Wafer-9371A.
- Power up the PC again, hit "DEL" to enter setup and do the same procedure as when booting Linux. The hard drive should be detected as primary master.
- Sometimes, it was experienced that the BIOS didn't detect the newly changed connections the first time, but this was solved by booting down the pc, power it off, and boot up again.

If Linux is something that has to be used frequently, and it is desired to keep both OSes available without all this rebooting, a boot loader that makes the user choose which operating system to boot during startup should be installed. An example of such a bootloader is *grub*, but how the QNX compatibility is for that bootloader is neither explored nor tested.

Appendix D

How to boot CyberBikePC in DOS

The CyberBikePC is capable of booting from USB-memory, which makes a straightforward option to connecting CDROM or floppy drives to test the pc with other OSes. Note: changes to the BIOS settings could inflict the ability to boot from USB devices. If such problems occur, refer to [IEI Technology Corp., 2006, chap. 4].

First, an USB flash memory has to be configured to look like a DOS boot diskette. This is the not so quite straightforward part of the task, but at last a simple solution was found:

1. Put an USB flash memory without any important content into an available USB connector on a PC running Windows XP.
2. Download `windows_enabler.zip` from [Hewitt, 2002], or from the CDROM (see Appendix A).
3. Unzip the file. Make sure all files ends up in the same folder. Start the application by double clicking on `Windows Enabler.exe`. An icon similar to the one just double clicked will show up in the system tray.
4. Left click on the icon in the system tray bar to activate the Windows Enabler application. The icon should now be showed with the text "On" in the foreground.
5. Open "My Computer" in XP, and right click on the icon representing the USB flash memory. Choose "format" in the drop down menu.
6. In the window that now appears, select check the "Create an MS-DOS startup disk" box. This option is by default not possible to use on a USB memory like this, but here Windows Enabler comes into play. Format the USB-drive.
7. Right click on the windows enabler system tray icon, and select "Quit".
8. Stop and unmount the USB storage medium by left clicking on its system tray icon.

To start the CyberBikePC from the USB drive, do the following:

1. Insert the bootable USB device into the USB-connector on the Wafer-9371A, via the "10-pin-to-USB" cable provided with the in the Wafer kit.
2. Power up the CyberBikePC. Hit "DEL" to enter BIOS setup.
3. Choose the "Advanced CMOS setup".
4. A menu similar to the one showed in Figure D.1 appears. Make sure the USB-device is booted from before any other devices (i.e. set "1st Boot Device" to USB and "2nd Boot Device" to IDE-0).
5. Save and exit BIOS setup, to restart system.

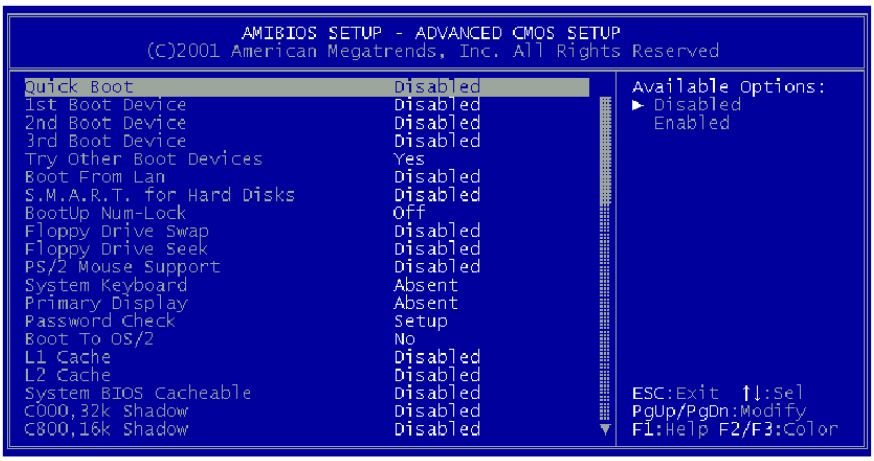

Figure D.1: Screenshot of the Advanced CMOS setup menu, taken from [IEI Technology Corp., 2006].

To get back into QNX, powering down the CyberBikePC, remove USB device, and power up again should be sufficient. If not, go into the BIOS setup again to make sure IDE-0 is specified as "1st Boot Device".

Appendix E

Connection tables

This appendix provides tables covering most of the connections in the CyberBike project. A lot of information is presented in more than one table, to make it easier to get an overview of connections while debugging, making additional connections, etc. This should be kept in mind when updating or changing the tables, to keep the documentation consistent, see Section 8.7.

The tables here is made in Microsoft Excel, and exported to LaTeX by an `excel2latex`-macro (see [Marder, 2001], and the enclosed CDROM, see Appendix A).

E.1 Terminal block outside suitcase

Legend: NO = Normally Open
NC = Normally Closed
B1 = left switch
B2 = right switch
BX - in = connected to DMM-32-AT
BX - NC-out; connected to positive side of diode
BX - NO-out; connected to negative side of diode

Lower terminals		Upper terminals	
Potmeter steering; signal	2	1	Tachometer propulsion
Tachometer steering	4	3	Potmeter pendulum; signal
	6	5	Tachometer pendulum
	8	7	
	10	9	
Tachometer propulsion; GND	12	11	
Tachometer pendulum; GND	14	13	Tacometer steering; GND
Pendulum switch; B2 - NO-out	16	15	Pendulum switch; B2 - NC-out
Pendulum switch; B1 - in	18	17	Pendulum switch; B2 - in
Pendulum switch; B1 - NO-out	20	19	Pendulum switch; B1 - NC-out
	22	21	
Potmeter pendulum; 5V	24	23	Potmeter steering; 5V
Potmeter pendulum; 0V	26	25	Potmeter steering; 0V

Table E.1: Terminal block outside suitcase.

Terminal	Connection
1	Tachometer propulsion
2	Potmeter steering; signal
3	Potmeter pendulum; signal
4	Tachometer steering
5	Tachometer pendulum
6	
7	
8	
9	
10	
11	
12	Tachometer propulsion; GND
13	Tacometer steering; GND
14	Tachometer pendulum; GND
15	Pendulum switch; B2 - NC-out
16	Pendulum switch; B2 - NO-out
17	Pendulum switch; B2 - in
18	Pendulum switch; B1 - in
19	Pendulum switch; B1 - NC-out
20	Pendulum switch; B1 - NO-out
21	
22	
23	Potmeter steering; 5V
24	Potmeter pendulum; 5V
25	Potmeter steering; 0V
26	Potmeter pendulum; 0V

Table E.2: Sorted two-column version of Table E.1.

E.2 Baldor

	Axis	Motor	Pin row - c			Pin row - a		
			Pin	Name	Connected to	Pin	Name	Connected to
Connector 2	3	Propulsion	2	Ref. Input	X2:5	2	Ref. Input	X2:6
	3	Propulsion	4	Tacho in	X1:1	4	Disable-input	EmergencyButton con2
	3	Propulsion	6	NC		6	Current monitor	
			8	NC		8	NC	
			10	NC		10	NC	
	3	Propulsion	12	Fault-out OC		12	NC	
	3	Propulsion	14	Ref. GND	X1:12	14	NC	
	3	Propulsion	16	+V DC	24V Battery/Power source	16	+V DC	
	3	Propulsion	18	Power Motor A1		18	Power Motor A1	Propulsion Motor +V
	3	Propulsion	20	Power Motor A2		20	Power Motor A2	Propulsion Motor -V
	3	Propulsion	22	0V DC	0V Battery/Power source	22	0V DC	
			24	NC		24	NC	
	2	Pendulum	26	Ref. input	X2:1	26	Ref. Input	X2:2
	2	Pendulum	28	Current monitor		28	Tacho in	
	2	Pendulum	30			30	Disable-input	EmergencyButton con2
	2	Pendulum	32	Fault out OC		32	Ref. GND	
Connector 1	2	Pendulum	2	+V DC		2	+V DC	24V Battery/Power source
	2	Pendulum	4	Motor A1		4	Power Motor A1	Pendulum Motor +V
	2	Pendulum	6	Motor A2		6	Power Motor A2	Pendulum Motor -V
	2	Pendulum	8	0V DC		8	0V DC	
	1	Steering	10	NC		10	Current monitor	
	1	Steering	12	Tacho in		12	Fault-out OC	
	1	Steering	14	Ref. Input	X2:3	14	Disable input	EmergencyButton con2
	1	Steering	16	Ref. GND		16	Ref. Input	X2:4
	1	Steering	18	+V DC		18	+V DC	24V Battery/Power source
	1	Steering	20	Power Motor A1		20	Power Motor A1	Steering motor +V
	1	Steering	22	Power Motor A2		22	Power Motor A2	Steering motor -V
	1	Steering	24	0V DC		24	0V DC	
	1	Steering	26	0V DC	0V Battery/Power source	26	0V DC	
	1	Steering	28	+V DC		28	+V DC	24V Battery/Power source
	1	Steering	30	Fault-out		30	Synchron-input	
	1	Steering	32	+14V / 50mA out	EmergencyButton con1	32	-14V / 50mA out	

Table E.3: Connection table for Baldor TFM 060-06-01-3

E.3 Terminal Blocks

Outside Card/unit	Pin	Block	Terminal	Inside Card/unit	Pin	Comment:
Propulsion tacho	Signal	X1	1	BALDOR	2:a4	Measurement signal for speed control on BALDOR card
Potmeter steering	Signal	X1	2	DMM-32-AT	J3:5	Input Channel 1
Potmeter pendulum	Signal	X1	3	DMM-32-AT	J3:7	Input Channel 2
Steer tacho	Signal	X1	4	DMM-32-AT	J3:9	Input Channel 3
Pendulum tacho	Signal	X1	5	DMM-32-AT	J3:11	Input Channel 4
		X1	6	DMM-32-AT	J3:13	Input Channel 5
		X1	7			
X1	1	X1	8	DMM-32-AT	J3:19	Connects propulsion tacho to I/O-Vin 8+ (differential input)
X1	12	X1	9	DMM-32-AT	J3:20	Connects propulsion tacho to I/O-Vin 8- (differential input)
		X1	10			
		X1	11			
Propulsion tacho	GND	X1	12	BALDOR	2:c14	Measurement ground for speed control on BALDOR card
Steer tacho	GND	X1	13	DMM-32-AT	J3:2	Agnd on I/O card
Pendulum tacho	GND	X1	14	DMM-32-AT	J3:2	Agnd on I/O card
Pendulum sw. B2	NC-out	X1	15	D2 / X2	+ / 2	Connected to positive side of Diode 1 and X2:2
Pendulum sw. B2	NO-out	X1	16	D2	-	Connected to positive side of Diode 2
Pendulum sw. B2	In	X1	17	X1	19	Series connection of the two switches
Pendulum sw. B1	In	X1	18	DMM-32-AT	J3:35	Vout 3 on I/O card
Pendulum sw. B1	NC-out	X1	19	X1	17	Series connection of the two switches
Pendulum sw. B1	NO-out	X1	20	D1	-	Connected to negative side of Diode 1
		X1	21			
		X1	22			
Potmeter steering	5V	X1	23	DMM-32-AT	J3:39	Vref. out on I/O card
Potmeter pendulum	5V	X1	24	DMM-32-AT	J3:39	Vref. out on I/O card
Potmeter steering	0V	X1	25	DMM-32-AT	J3:2	Agnd on I/O card
Potmeter pendulum	0V	X1	26	DMM-32-AT	J3:2	Agnd on I/O card
BALDOR	2:c26	X2	1	DMM-32-AT	J3:1	Connects I/O-Agnd to pendmotor Vref-
BALDOR	2:a26	X2	2	D2 / X1	- / 15	Connects I/O-Vout 3 (da3) to pendmotor Vref+ (via switches)
BALDOR	1:c14	X2	3	DMM-32-AT	J3:1	Connects I/O-Agnd to steermotor Vref-
BALDOR	1:a16	X2	4	DMM-32-AT	J3:37	Connects I/O-Vout 1 (da1) to steermotor Vref+
BALDOR	2:b2	X2	5	DMM-32-AT	J3:1	Connects I/O-Agnd to propmotor Vref-
BALDOR	2:a2	X2	6	DMM-32-AT	J3:38	Connects I/O-Vout 0 (da0) to propmotor Vref+

Table E.4: Connection table for the terminal blocks X1 and X2.

E.4 J3 on DMM-32-AT

On DMM-32-AT		Connected to		Further connected to		Comment
Pin	Name	Block	Terminal	Card/unit	Pin	
1	Agnd	X1	17	Pendulum switch B2	In	
1	Agnd	X2	3	BALDOR	1:c14	Vref - steermotor
1	Agnd	X2	5	BALDOR	2:b2	Vref - propulsion motor
2	Agnd	X1	13	Tacometer steering	GND	
2	Agnd	X1	14	Tachometer pendulum	GND	
2	Agnd	X1	25	Potmeter steering	0V	
2	Agnd	X1	26	Potmeter pendulum	0V	
5	Input Channel 1	X1	2	Potmeter steering	Signal	
7	Input Channel 2	X1	3	Potmeter pendulum	Signal	
9	Input Channel 3	X1	4	Tachometer steering	Signal	
11	Input Channel 4	X1	5	Tachometer pendulum	Signal	
13	Input Channel 5	X1	6			Easy Available
19	Differential Input Channel 8+	X1	8	X1	1	Propulsion tacho +
20	Differential Input Channel 8-	X1	9	X1	12	Propulsion tacho -
35	Vout 3	X1	18	Pendulum switch B1	In	
37	Vout 1	X2	4	BALDOR	1:a16	Vref + steermotor
38	Vout 0	X2	6	BALDOR	2:a2	Vref + propulsion motor
39	Vref. out	X1	23	Potmeter steering	5V	
39	Vref. out	X1	24	Potmeter pendulum	5V	

Table E.5: Connection table for J3 on DMM-32-AT.

Appendix F

Design of PCB

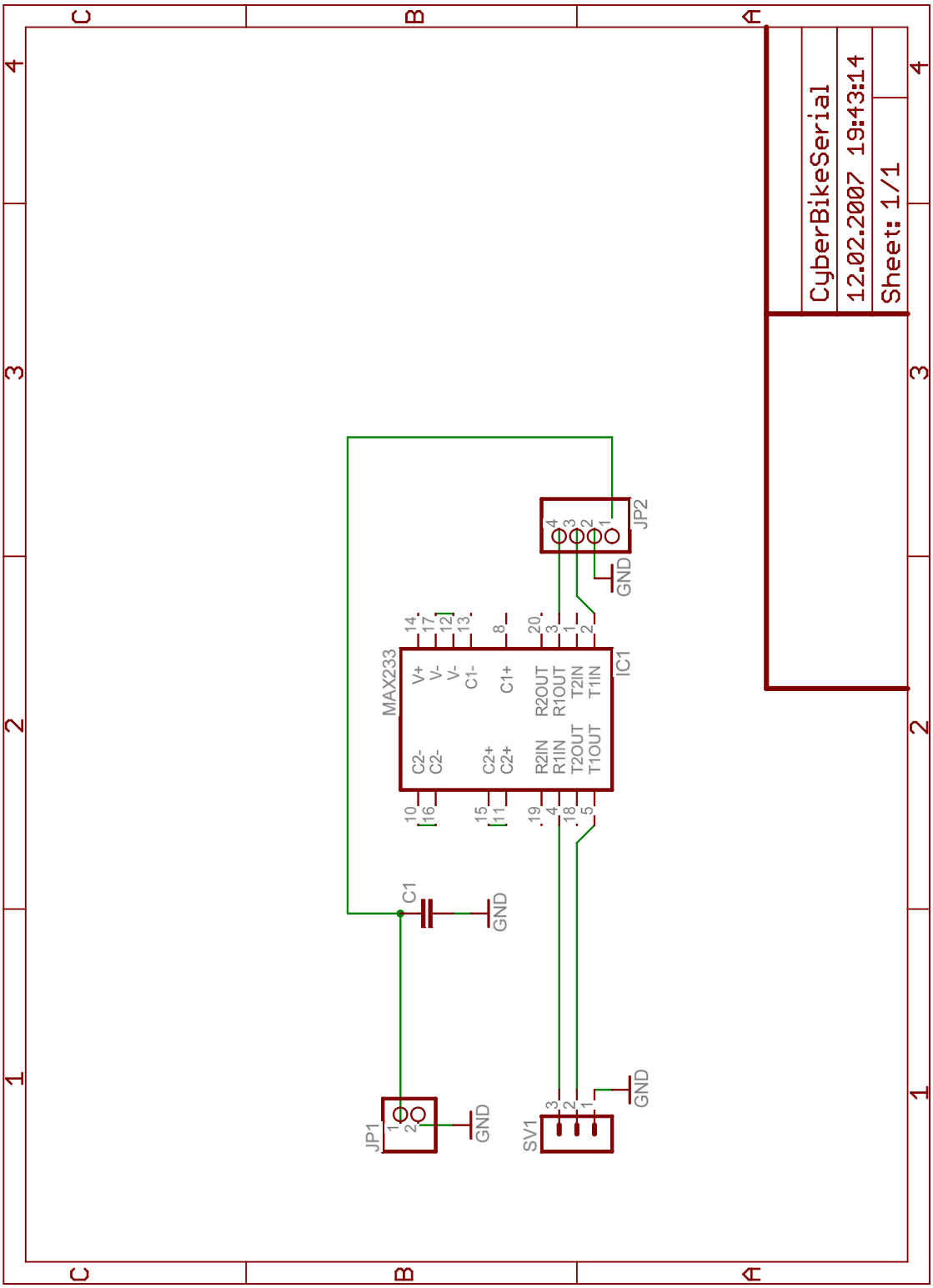

Figure F.1: Schematic design of PCB.

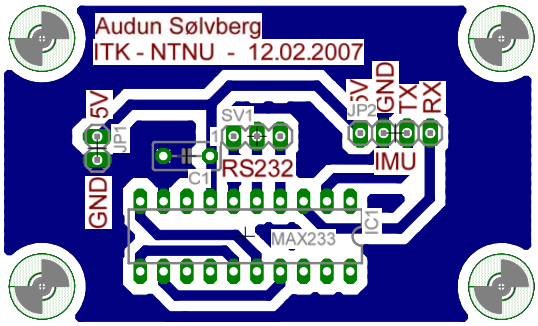

Figure F.2: Layout of PCB, all layers showed.

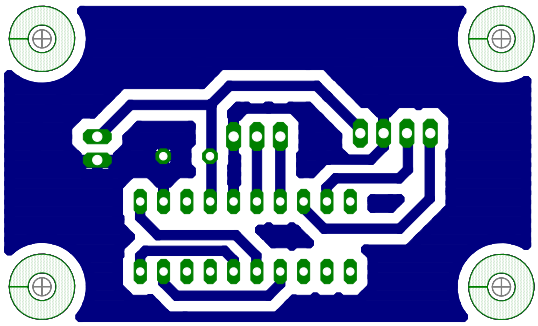

Figure F.3: Layout of PCB, bottom surface print

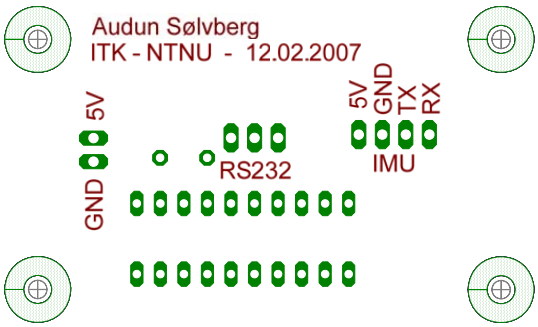

Figure F.4: Layout of PCB, top surface print.