Norwegian University of
Science and Technology

Autonomous Bicycle

Snorre Eskeland Brekke

Master of Science in Engineering Cybernetics
Submission date: September 2010
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, which has been developed by the department's students during the past few years.

Work that has to be done reach this goal involved:
-software development with real-time requirements
-use of a real-time operating system
-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: 25. April 2010
Supervisor: Amund Skavhaug, ITK

Autonomous Bicycle

Master's Thesis

Snorre Eskeland Brekke
snorrees@stud.ntnu.no

Engineering Cybernetics
Embedded Systems

Supervisor:
Amund Skavhaug

Hand in date: September 8, 2010

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

Preface

As I am handing over this thesis for evaluation, I am inspired by the knowledge that the autonomous bike will have two new students continuing the work I now leave behind. The project, a daunting task in its own right, is in my opinion the epitome of Engineering Cybernetics, revealing the field in all its gore and glory. Personally, I had never really contemplated how varied Cybernetics could be.

Throughout the work of the thesis, I stepped out of my comfort zone, out of the software domain, and into the comparatively unreliable world governed by hardware. Up until this project, the very idea of even probing electronics was very foreign and unfamiliar terrain. I have never, unlike most of my fellow peers, built a computer, and the bike system is so much more than a mere workstation. The autonomous bike is responsible for teaching me a methodical approach to problems, breaking them down to the basics, and solving them step by step. I have also surprised myself as to what can be accomplished alone, given sufficient time.

I am torn by the thought of stopping work on the bike. On one hand, the project has been a lot of fun, both challenging and entertaining. On the other hand, grueling, unexplained system failure, very much following Murphy's Law will not exactly be missed.

I would like to thank and give a loud shoutout to my supervisor Amund Skavhaug, whose understanding and guidance has served as a rock for the duration. Without his continued encouragement and patience, this thesis, and indeed my master's degree, might not have come to be. I would also like to thank the guys at the workshop, dealing with my timid questions even as they missed their bus ride home.

Finally I would like to thank my girlfriend Sara, for simply being there through it all. Your nuanced view on life is indeed an inspiration to a square like myself.

Trondheim, September 8, 2010

Snorre Eskeland Brekke

Contents

Abbreviations and acronyms **vii**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Problem . . . 1
1.3 Earlier work . . . 2
1.4 The thesis structure . . . 2

2 Background **3**
2.1 The Bike model . . . 3
2.1.1 Important assumptions and simplifications . . . 3
2.1.2 Modelling the different parts . . . 5
2.1.3 Other important parameters . . . 6
2.2 QNX Neutrino . . . 11
2.2.1 Message Passing . . . 11
2.2.2 Resource Managers . . . 13
2.3 Real-time Workshop . . . 17
2.3.1 Configuration . . . 17
2.3.2 S-functions . . . 18
2.3.3 Communication . . . 20
2.3.4 Compiling and Running the Code . . . 21
2.4 System flow . . . 21

3 Hardware **23**
3.1 Current source . . . 23
3.2 Computer . . . 23
3.3 Storage . . . 27
3.4 I/O card . . . 27
3.4.1 Motor controller card . . . 28
3.5 Motors . . . 29
3.5.1 Propulsion motor . . . 30
3.5.2 Motor for the inverted pendulum . . . 30
3.5.3 Steering motor . . . 31

3.6 IMU - Xsens (MTi) 32
3.7 IMU - Spark Fun 32
3.8 GPS 33
3.9 Emergency Stop 33
3.10 Fan 35
3.11 Batteries 35
3.12 Pendulum Limit Switches 35
3.13 Potmeters 35

4 Software **37**
4.1 Software 37
4.2 OS[1] 37
4.3 Drivers 37
4.3.1 Resource Manager: `devc-dmm32at` 39
4.3.2 Resource Manager: `devc-imu` 40
4.3.3 Resource Manager: `devc-mt` 42
4.3.4 Resource Manager: `devc-gps` 43
4.3.5 Resource Manager: `devc-velo` 43
4.3.6 Simulink Bike Demo 44

5 Problems and Solution **47**
5.1 New Motherboard 47
5.2 New Harddrive 49
5.3 OS Update 50
5.4 The Cyberbike Model 51
5.5 Bike Control Demo 52
5.6 Driver Enhancements 58
5.6.1 The Velo driver 58
5.7 Physical Challenges and Enhancements 58
5.7.1 Cabinet Frame 58
5.7.2 Potmeter strain 59
5.7.3 Pendulum Limit Switches 60
5.8 The Propulsion motor 60
5.9 Wiring issues 60
5.10 The MTi 60
5.11 The Wireless Issue 61

6 Tests and Experiments **63**
6.1 Test equipment 63
6.2 The Wafer-945GSE2 63
6.2.1 USB Ports 63
6.2.2 COM Ports 63

[1] Operating System

6.2.3 Network Issues 64
6.2.4 Harddrive Issues 65
6.3 Motor tests 65
6.4 Batteries stress test 65
6.5 Simulink tests 65
6.5.1 Cyberbike model tests 66
6.5.2 Bike demo tests 66
6.6 Device Peripherals and associated drivers 66
6.7 GPS 68

7 Further Work 69
7.1 Recap of Existing Problems 69
7.1.1 The first journey 69
7.1.2 Minor Issues 69
7.1.3 Wireless networking 71
7.2 Moving Forward 71
7.2.1 Simulink model 71
7.2.2 Optimizing `devc-dmm32at` 72
7.2.3 Videocamera 72
7.2.4 Mechanical Brakes and Overall Safety 72
7.2.5 Overall system 73

8 Discussion 75
8.1 Choice of new motherboard 75
8.2 Choice of storage medium 75
8.3 Unresolved problems 76
8.3.1 Lack of Wireless Networking 76
8.4 Reflection 77

9 Conclusion 79

Bibliography 81

A Contents on DVD 83

B How to start the bike system 85

C Connection tables 89
C.1 Terminal block outside suitcase 90
C.2 Baldor 93
C.3 Terminal Blocks 95
C.4 J3 on DMM-32-AT 97

Abbreviations and acronyms

A/D Analog-to-Digital

AC Alternating current

API Application programming interface

BIOS Basic Input/Output System

DAC Digital-to-Analog Converter

DC Direct Current

DHCP Dynamic Host Configuration Protocol

DMA Direct Memory Access

EIDE Enhanced IDE (Integrated Drive Electronics, se *ATA*)

FIFO First In - First Out

GUI Graphical User Interface

I/O Input/Output

IOV I/O vector

IDE Integrated Development Environment

IMU Inertial Measurement Unit

IPC Inter Process Communication

LQG Linear quadratic Gaussian (control)

OS Operating System

PID Proportional-Integral-Derivative (controller)

POSIX Portable Operating System Interface
'X' refers to the Unix heritage of POSIX

QNX A RTOS developed by QSSL

QSSL QNX Software Systems Limited

RM Resource Manager
also refered to as "device drivers"

RTOS Real-Time Operating System

RTW Real-Time Workshop

TLC Target Language Compiler

TCP/IP Transmission Control Protocol/Internet Protocol

SATA Serial Advanced Technology Attachment

SBC Single-board computer

UART Universal Asynchronous Receiver Transmitter

USB Universal Serial Bus

μC micro controller

Abstract

The autonomous bike was conceived by Jens G. Balchen back in the eighties, and later picked up by Amund Skavhaug. The idea of a two-wheeled, self-powered, yet riderless bike has since been pursued intermittently over the years. Audun Sølvberg was the last person working on the project in 2007, making large headway towards the final goal of the bike: An outdoor ride, preformed by a riderless bicycle.

The same goal is shared by this thesis; to make the bike work as intended and get the bicycle running outside. Additionally, as it would be naive to assume that this work would leave the bike in perfect condition, the thesis focuses on documenting issues that are left unfixed upon completion. Unfortunately, a fully functional, outdoor demonstration was not achieved, but half of the battle was won: Two of the three motors residing on the bike can be fully controlled by an external device.

At the start of the work, the instrumentation system was mostly completed. A computer, running a QNX Neutrino Operating System, interfacing with potmeters, motors, Inertial Measurement Units and a GPS where available, mounted on the bike. Drivers communicating with the hardware was already written, and a Simulink model, meant to control the bike had been developed. However, the motherboard was in need of replacement, and the Simulink model was not finalized or even tested. The system was also lacking wireless networking capabilities.Yet, at the onset of the thesis, it was believed that the project was very close to reaching its ultimate goal.

As the work progressed, several issues became apparent, emerging along with problems being solved. The motherboard was replaced with a Wafer-945GSE2, implicitly requiring a new hard drive to be installed. The OS was upgraded from version 6.3.0 to 6.4.1, to bring it up to date and as a requirement for some of the motherboard hardware. All of the device drivers where modified to work without their counterpart hardware connected, easing development of the Simulink model. The model was shown to be unreliable, but the hardware interface subsystem was completed and tested to allow for easy integration in separate projects.

To demonstrate the capabilities of subsystem, a Bike Demo model was created, allowing the steer and pendulum to be controlled by the used of the bike accelerometer, using orientation data as reference. The intent was for the demo to serve as a control system for an outdoor ride. Ultimately the motor controller card was found to be incompatible with the propulsion motor, unable to deliver sufficient current. At the time of discovery, too little time reminded for the problem to be rectified. A demonstration of control system however, was successfully concluded.

An attempt to bring wireless networking to the bike failed. Arguably too much time was spent making USB Wifi devices working with the QNX OS, when easier alternatives could perhaps have solved the problem earlier.

The thesis has a large focus on the work ahead, as the system is complex and unreliable by its very nature. All known issues are detailed and summarized, and the various chapters describing hardware and software have been outlined in an attempt to serve as a go-to reference for students taking on the autonomous bike project in the future.

Chapter 1

Introduction

1.1 Motivation

The autonomous bike was conceived by Jens G. Balchen back in the eighties. He imagined the construction of a two-wheeled, autonomously running bicycle, an idea picked up and future developed by Amund Skavhaug. Since its conception, the autonomous bike project has intermittently been the center of attention, slowly nearing completion. The goal of this master thesis has therefor been twofold: to take the bike on a ride outdoors for the first time, and prepare the bike for future development, documenting problems as they appear.

The Department of Engineering Cybernetics can conceivably use the bike as a advert and poster child, attracting potential students or as an exhibit for visitors. The bike covers a wide variety of subjects relevant to Cybernetics, and will serve as an excellent example of what can be achieved throughout the studies. The theory backing the bike dynamics is also of interest, as presumably no bicycle has ever been controlled in this manner: an inverted pendulum for balance, a motor for controlling the steering direction, and a motor providing propulsion.

1.2 Problem

The main goal is to get the bike running. The instrumentation system is mostly complete, but a new motherboard and damaged weiring has to be repaired. Making accommodations for wireless networking is necessary. Furthermore, the Simulink model responsible for controlling the bike has to be completed, as it is not operational. Based upon earlier work, it is also assumed that an instrumentation system of this magnitude will present a host of unforeseen problems, all of which will have to be dealt with as they appear. As such, another important aspect of the thesis is documenting issues and uncompleted tasks so that the work can be easily continued in the future.

This thesis, along with the work done to get the bike on the road, can hopefully be used as a point of reference and documentation for the system. The autonomous

bike is a relatively complicated instrumentation system, and unifying previous work and documentation should be prioritized.

1.3 Earlier work

This thesis is based upon the fall project of the candidate. As the project and the master thesis is part of an integral process, the project report will not be referenced, but instead be used directly. The most important previous work is the master thesis of [Sølvberg, 2007], who came close to conducting an outdoor ride. [Sølvberg, 2007] enhanced the instrumentation systems already implemented on the bike, and updated the software. He installed a GPS-module and made a new and improved accelerometer available for the bike. [Sølvberg, 2007] based most of his work on the theses of Loftum [2006] and Bjermeland [2006]. Bjermeland [2006] developed a mathematical model for the bike, and implemented a Simulink model and simulator. Bjermeland [2006] developed the bike instrumentation system, and was responsible for installing the bike computer running QNX. Fossum [2006] continued the work of Bjermeland [2006] , improving the instrumentation system further.

1.4 The thesis structure

The layout of this thesis has been done for it to function as a "go to" reference for future work. Combining the work of the previous theses, and putting relevant material in comprehensive chapters has been important. The background chapter focuses on the underlying theory backing up the bike system. The hardware chapter describes the instrumentation system, giving a broad overview of the components composing the system. The software chapter is meant as a reference guide for new users, explaining their use and functioning. Both of these chapters present the system as is, at the end of the thesis work. Chapter 5 covers the practical work preformed during the thesis, outlining problems faced and solutions to them, followed by a chapter detailing tests and experiments performed during the thesis work. Chapter 7, detailing further work, is put *before* the conclusion, and indeed, before the discussion. The reason for this is simple; the tasks that remain for the autonomous bike are numerous, and detailing what needs to be done is as much part of the documentation as anything else. Therefore it is natural to include it as a part of the main text.

Chapter 2

Background

This chapter describes theory theory and systems upon which the autonomous bike is based. The mathematical model and the computer software which runs on the bike computer work in unison, and to get an understanding of how this is accomplished, several terms and principles will be presented. Finally a broad picture of how the total system works in terms of signal flow is presented.

2.1 The Bike model

Bjermeland [2006] developed a mathematical model of the bike, as well as implementing a control algorithm which was tested using simulation. To get an understanding of how the bike is supposed to operate, it is therefore important to understand some of the underlying theory. Bjermeland [2006] developed the model, controller and simulator for the bike using steering and leaning (the inverted pendulum) as the manipulated variables. This summary is in large based upon [Sølvberg, 2007], and the background presented there. [Lebedonko, 2009] also made a tremendous effort detailing theory and modelling of a bike. His work will not be presented here, as it has not been implemented into the actual system in any way. His work, however, will possibly become important moving forward.

2.1.1 Important assumptions and simplifications

To ease the modelling of the bike, some assumptions and simplifications where made:

The bike was first 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 the simplified model of the bike, where the centers of mass is indicated. Elasticity and other non-ideal movements and deformations is neglected.

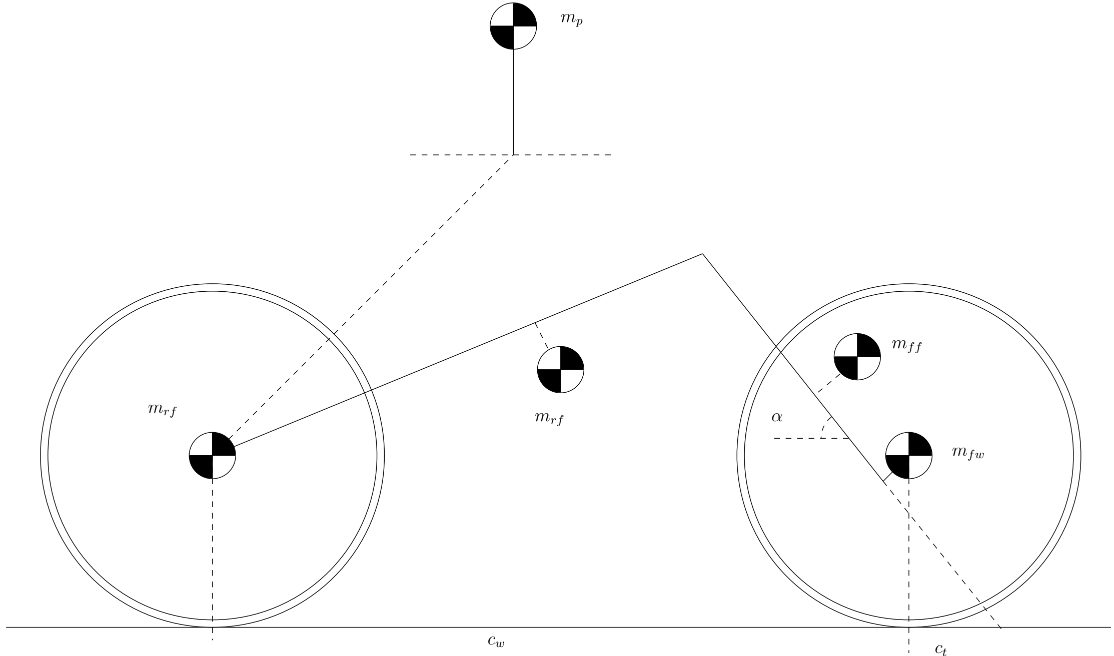

Figure 2.1: The bike main parameters. Figure from [Sølvberg, 2007].

The bike orientation is space is shown in Figure 2.2; the Z-axis points upwards and the X-axis points in the bike direction of travel when the yaw-angle ψ is zero.

A rotational matrix can be approximated for small angles as shown in Equation (2.1):

$$\begin{aligned}
R_b^a &= R_z(\psi)R_y(\theta)R_x(\phi) \\
&= \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix} \\
&= \begin{bmatrix} \cos(\psi)\cos(\theta) & -\sin(\psi)\cos(\phi) + \cos(\psi)\sin(\theta)\sin(\psi) & \sin(\psi)\sin(\phi) + \cos(\psi)\cos(\phi)\sin(\theta) \\ \sin(\psi)\cos(\theta) & \cos(\psi)\cos(\phi) + \sin(\phi)\sin(\theta)\sin(\psi) & -\cos(\psi)\sin(\phi) + \sin(\theta)\sin(\psi)\cos(\phi) \\ -\sin(\theta) & \cos(\theta)\sin(\phi) & \cos(\theta)\cos(\phi) \end{bmatrix} \\
&\approx \begin{bmatrix} 1 & -\psi & \theta \\ \psi & 1 & -\phi \\ -\theta & \phi & 1 \end{bmatrix}
\end{aligned} \quad (2.$$

This is a linear rotational matrix. Note that this model will produce large errors when the angles become wider.

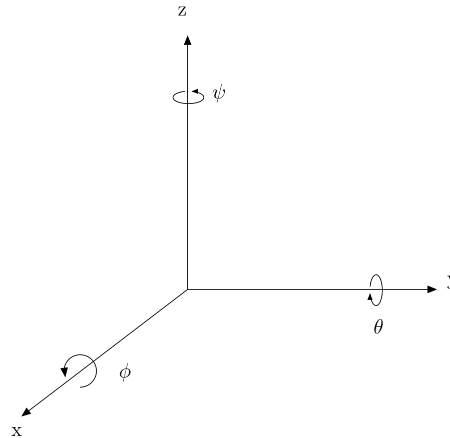

Figure 2.2: Space orientation in a right hand coordinate system.

Another important assumption is that the bike is symmetrical lengthwise while in equilibrium. For the autonomous bike this does not hold true , as the steering motor is placed on the left side of the frame. However, the batteries are heavy compared to most of the bike elements and contributes greatly to the rear frame center of mass, somewhat reducing the effect of the motor placement.

Furthermore; it is assumed, as a consequence of the linearized equations, that the bike speed is kept constant. This is a problem in a real world scenario where keeping constant speed can be hard to achieve. This comes in addition to the fact that the pitch-angle is assumed to be zero at all times. In other words, the bike model is meant for movement on a flat surface.

2.1.2 Modelling the different parts

The position and parameters of the five rigid bodies can be described with with rotation and transformation matrices relative to the inertial frame i. Connecting the rear wheel and the rear frame, the front wheel and the front frame reduces the total number of mass centers from five to three. The posision of these centers of mass is then given:

Rear frame and rear wheel m_r:

$$r^i_{m_r} = \begin{bmatrix} x_{rw} + x_r \\ y_{rw} + x_r\psi_r - z_r\phi_r \\ z_r \end{bmatrix} \tag{2.2}$$

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 get an understanding of how the Simulink model is connected to the actual bike, some important variables needs to be explained. Two important vectors used 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 set turnrate for the bike. This is measured or estimated through an observer as described in [Bjermeland, 2006, chap. 6]. A LQG[1] controller is used, making it possible to weigh the different error stats in the physical system, and penalize excessive actuator use. The observer states are shown in Equation (2.7) and (2.8), and the state space model is described by Equation (2.10) and 2.11).

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

$$\begin{aligned} x_2 &= \dot{x_1} \\ &= \begin{bmatrix} \dot{\phi}_r \\ \dot{\delta} \\ \dot{\phi}_p \end{bmatrix} \end{aligned} \tag{2.8}$$

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

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

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

where

- M is the 3×3 *Mass Matrix*
- $K0$ is the 3×3 *Velocity Independent Stiffness Matrix*

[1] Linear quadratic Gaussian (control)

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

[Bjermeland, 2006] gives a much more throughout explanation of the different parameters, as a full coverage here is beyond the scope of the thesis.

The Kalman fliter is based on Equation (2.11). In the model it is assumed that the turning rate of the bike is estimated through the measurement of the steering angle. On the actual bike implementation, the gyro is used to measure yaw, so this has to be considered when working on the actual bike.

Matlab and Simulink model

The Simulink model was initially developed by [Bjermeland, 2006] and then future enhanced by [Sølvberg, 2007]. A Matlab S-function reads data from the `devc-velo` resource manager, and transmits this data to the control block. The model is shown in Figure 2.3.

The model is dependent upon several files to work as intended. These are shown in Table 2.1, and are automatically loaded when the Simulink model is initialized.

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

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

A GUI[2] is displayed when the simulation is completed or stopped, as shown in Figure 2.4 .

A Runge-Kutta fixed step solver is used for the model. Section 2.3 will cover how the bike can be run and communicate with hardware.

[2]Graphical User Interface

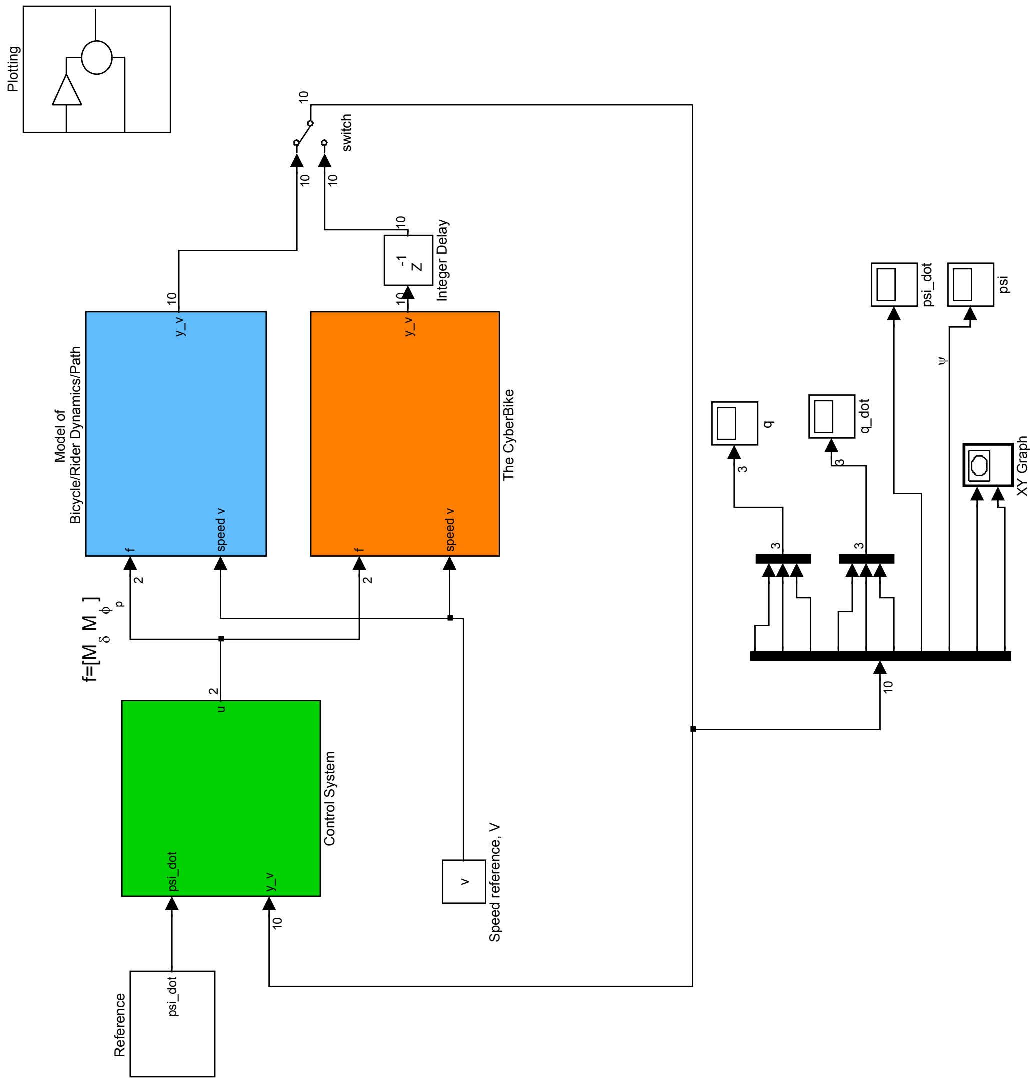

Figure 2.3: The main blocks in the CyberBike Simulink model.

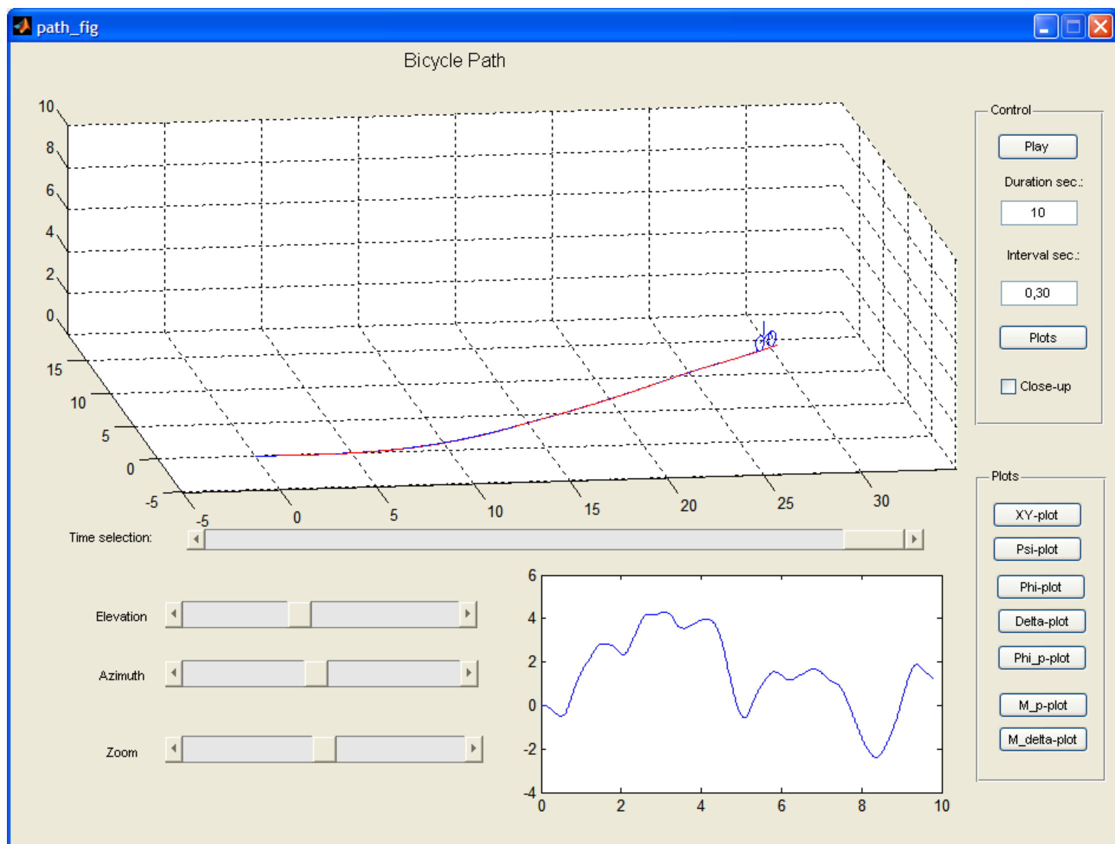

Figure 2.4: Screenshot of the simulation GUI, constructed by Bjermeland [2006].

2.2 QNX Neutrino

QNX[3] Neutrino is a RTOS[4] developed by QSSL[5]. QNX has a micro kernel architecture; only the most fundamental services are run by the kernel. Beyond simple primitives and system calls to assign runtime, address space and process communication, all services run in user space. These programs are typically called *servers*. The operating system is therefor completely modular, and the user can start and stop servers as needed. As the modules do not share memory with the kernel - QNX has protected memory - servers that hangs will not crash the system. This has great impact on systems where system failure can lead to damaged equipment or bodily harm.

QNX Neutrino relies on two important concepts, the first being *message passing* and the second being *resource managers*. Both will be explained in the following sections, based upon [QNX Software Systems, 2009b, Kap. 3] and [QNX Software Systems, 2009a, Kap. 4].

2.2.1 Message Passing

IPC[6] or interprocess communication, is a major part of what transforms QNX Neutrino from being an embedded realtime kernel to a fully fledged POSIX[7] operating system. Modules can communicate through messages, and the architecture ensures that IPC on separate, distributed nodes can seamlessly be included. This is a major advantage over traditional systems, where local and external communication is treated very differently. Through the Neutrino C library, which handles the message passing, POSIX C calls can be used without having to write separate functions for message handling.

Message passing uses a server/client model. A server awaits client requests, which in turn are blocked until the server responds. Under Neutrino, the delivery of messages is done using interconnected channels, and are not directed to one particular thread. This is a strength in that only one kernel object is need for all threads that are connected to a given channel, which in turn increases efficiency. Another advantage is that when a threads becomes blocked, the receiving thread is immediately readied to run, without any explicit work being done by the kernel, something which is common for other types of IPC.

A server will first create a channel using the *ChannelCreate()* function. Clients can then connect to the channel by calling *ConnectAttach()*, which returns a *connection ID (coid)*. The ID is used by the client to send messaged to the server with the *MsgSend()* function. The server is blocked through *MsgReceive()*, a function call that awaits request sent by *MsgSend()*. When the server has handled the request, a response is sent using *MsgReply()*, upon which the client can continue. *MsgReply()* does not block the server,

[3] A RTOS developed by QSSL
[4] Real-Time Operating System
[5] QNX Software Systems Limited
[6] Inter Process Communication
[7] Portable Operating System Interface

as it is implied that the client is already awaiting an answer from the server. If for whatever reason the request was denied, the server will replay using *MsgError()*, giving the client some information about the failed request.

This type of communication is synchronised, and as such a strict message hierarchy has to be adhered to. This is important to avoid deadlocks, where two servers ends up waiting for each others response. Two threads should never send messages to each other; all cross communication should come in the form of replies from the server. In a similar fashion, *MsgSend()* should only be used by threads on a higher level in the hierarchy as shown in Figure 2.5.

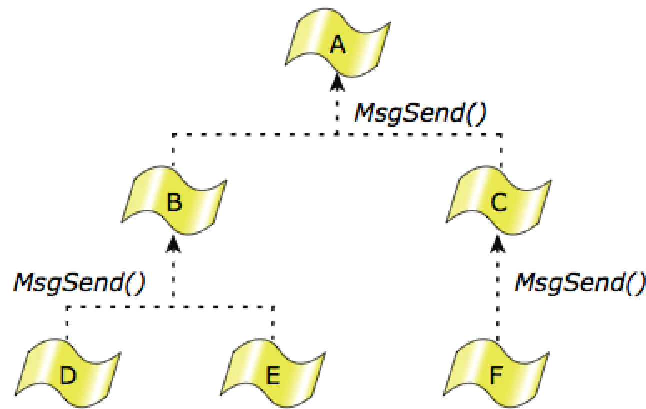

Figure 2.5: Communication hierarchy, taken from [QNX Software Systems, 2009b] .

If greater flexibility beyond the protocol presented thus far is needed, QNX Neutrino provides the *MsgDeliverEvent()* function. This is a asynchronous, non-blocking function call. It is useful when a client wants a server to initiate a time consuming process, but does not want to be blocked while the server is working. The function also allows threads on a higher level to message threads lower in the hierarchy about timers, interrupts and other events, independent of client requests.

Another, non-blocking option, is *pulses*. When *MsgReceive()* returns 0 as receiver ID, this indicates a pulse. A pulse is limited by 40 bits of information, thus limiting the use.

QNX messages are transferred by copying the message directly from one memory address space to another. By doing so, extra buffers to transfer the messages are avoided, and the kernel can preform the transfer without the need for additional meta information. This has the benefit of low overhead and high bandwidth, close to that of the underlying hardware. If a message is larger than the server is can receive, the server can use the *MsgRead()* function to read the rest of the data, directly from client memory. Similarly, the server can use *MsgWrite()* to transfer large messages directly to client memory, should it be necessary. This is particularly useful when transferring large amounts of information, and avoids large buffers to handle large requests.

The message passing primitives also supports transfers with several parts. The parts need not be a continuous memory space, but can be specified with a table which indicates

where the message fragments can be located (see Figure 2.6). IOV[8] is used to assemble such a message, and is sent similarly to single part messages, using functions with a v appended to the function name. In other words, *MsgWritev()* replaces *MsgWrite()* and so forth.

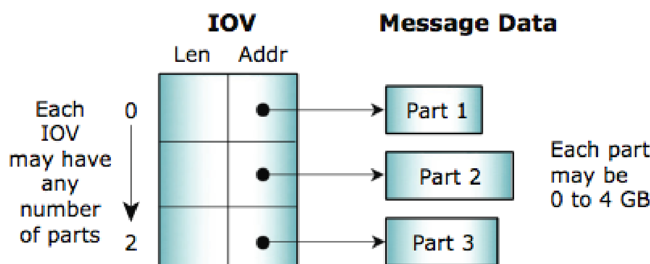

Figure 2.6: Data structure for multi part message . From [QNX Software Systems, 2009b] .

As mentioned earlier, these function calls are not needed when programming standard POSIX C, as the underlying library handles the actual message passing. Knowledge about QNX message passing, however, is a important to understand the system architecture, and plays a major part in the basis for the bicycle drivers: resource managers.

Qnet

Another feature of QNX is Qnet. In a network with all QNX Neutrion nodes, an application can access any resource on the network as if it was available locally. It is even possible to start programs on other nodes. The Qnet protocol makes the network solution for QNX appear transparent, by the use of message passing. The protocol is meant for protected networks, and provides little in the way of network security. Nodes on the network appear in the `/net` directory, listed by their host name. Section 5.3 describes how Qnet was configured for the bike system.

2.2.2 Resource Managers

Resource managers, or device drivers, are programs running in user space as servers. They receive messages from other programs and can communicate with hardware. As a process, a resource manager registers a path name prefix in the system which other processes can open with standard C *open()* or *fopen()* calls. Clients can then use *read()*, *write()* or other calls using file pointers and file descriptors. Resource managers are generally used as an interface to other devices. In other words, device drivers run in user space on QNX - separating it from typical operating systems - and can be started and stopped in real-time.

[8] I/O vector

There are several strengths to this implementation. Resource managers have POSIX API[9] and adheres to the documentation from the bottom up. This way, the learning curve for new developers is reduced. The number of interfaces is also low, as all server processes has the same form. This has the added benefit of making it easy for clients to connect to several different servers: all that is needed is *open()*, *read()* og *write()*. Last, but not least, resource managers are very easy to test. Command-line tools can be used to access a newly developed manager, without any additional test application. For instance, to test a resource manager that has registered the pathname `/dev/server`, a simple use of `cat /dev/server` would print what a *read()* call would have received. To test the *write()* function, using, for instance, `echo 50 >/dev/server` would send the value 50 to the manager as if a *write()* with 50 as an argument had been called.

Beneath the surface of the resource managers, QNX uses message passing for communication. The manager has a *MsgReceive()* loop answering any *MsgSend()* from external clients. A typical request is shown in Code sample 2.3.

Code sample 2.1 Typical request to a resource manager with the filepath `/dev/server` registered.

```
/* In this stage, the client talks to the process manager and the resource manager.*/
fd = open("/dev/server", O RDWR);

/*In this stage, the client talks directly to the resource manager. */
for (packet = 0; packet < npackets; packet++)
    write(fd, packets[packet], PACKET SIZE);
close(fd);
```

First the client calls *open()*, which in turn asks the process manager to return data associated with the given file path. If a resource manager has registered the path, the process manager is responsible for keeping track of it, and provide the necessary information. The client library proceeds with a connection message to the resource manager, using a *ConnectAttach()* call. The file descriptor which is returned by the call is then used to send messages directly to the manager, and validates if the call is valid (in the case of write protection or similar). Finally the *open()* call returns the file descriptor. The descriptor is used as an argument to all *read()* and *write()* calls to the resource manager.

Code sample 2.3 shows pseudo code for a typical resource manager, and all of the drivers on the bike computer is build using a similar principle.

Note that when using the library available in QNX, it is not necessary to explicitly call *MsgReceive()*. Instead serveral functions are available to ease the implementation. The most important functions are:

dispacth_ create() creates a dispatch struct which blocks when receiving messages.

[9] Application programming interface

Code sample 2.2 Pseudo code for a typical resource manager.

```
initialize the resource manager
register the path with the process manager
DO forever
    receive message
    SWITCH message type
        CASE _IO_CONNECT:
            call io_open handler
            ENDCASE
        CASE _IO_READ:
            call io_read handler
            ENDCASE
        CASE _IO_WRITE:
            call io_open handler
            ENDCASE
        /*other file calls are possible as well*/
    END SWITCH
END DO
```

.

iofunc_ attr_ init() initializes the attribute struct used by the device.

iofunc_ func_ init() initializes two data strucs, `cfuncs` and `ifuncs`, containing pointers to connect- and I/O[10]-functions, accordingly.

resmgr_ attach() creates the channel used by the resource manager to receive messages, and indicates to the process manager which path the resource manager should be responsible for. During this call, the three strucs mentioned above are connected, and the channel ID is created.

dispatch_ context_ alloc() allocates the internal context for the resource manager, which is used to process messages.

dispatch_ block() the function call which will run while the resource manager is blocking, and awaiting client requests.

dispatch_ handler() the function call which will run when a message is received and processes the request.

When the resource manager is started, there is typically only a small segment of code that runs continuously, as shown in Code sample 2.3.

When combined with the client example above, it is this part of the code which processes *open()*, *read()* and *write()* from the client. *dispatch_ block()* receives a messIage, and transfers it to the *dispatch_ handler()* function, which processes the message

[10]Input/Output

Code sample 2.3 Typical blocking loop run by a resource manger.

```
/* start the resource manager message loop */
while(1) {
    if((ctp = dispatch_block(ctp)) == NULL) {
        fprintf(stderr, "block error\n");
        return EXIT FAILURE;
    }
    dispatch_handler(ctp);
}
```

.

depending on the message type. The array with connect- and I/O-functions is then used to determine where the final processing is done. Finally the *dispatch_handler()* returns, and the resource manager goes back to the blocking routine until another message is received.

The behaviour of any resource manager is ultimately the implementation of the connect- and I/O-functions. These are determined by *iofunc_func_init()*. Additional functionality can be achieved by overwriting the standard functions with customized functions. The easiest way to do this is to create a new function that calls *iofunc_ *_ default()* ('*' can be *open, read, write* or *devctl()*), as part of the code. This ensures that the framework is kept intact, as well as additional code can be added.

The *devctl()* function is a general way to communicate with a resource manager. Clients can send data to, receive data from a resource manager, or even both, using *devctl()*) . For the bike drivers *devctl()*) plays a central role, and is therefor covered in more detail here. The function name is a short for *DEVice ConTroL*, and requires 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[11].

The command sent to *devctl()* is constructed by macros defined in `<devctl.h>`. The command word indicates the direction of data transfer, as well as the size of the data struct being sent. This way messages of the same class but with different datatypes can be identified, and the manager can handle several different type of messages, all using the same function.

[11] Resource Manager

The bike has the following drives running their own versions of *devctl()*: `devc-dmm32at`, `devc-velo`, `devc-imu`, `devc-gps` and `devc-mt`.

2.3 Real-time Workshop

To connect hardware and drivers on the bike to the Simulink model, RTW[12] is needed. RTW is an addition to Simulink which generates C-code and Makefiles based on Simulink models ([The MathWorks, Inc., 2009]). It is a powerful tool for rapid software development, as complex models easily can be created graphically and then compiled to run on very different hardware. Additionally, Simulink can be used as a tool to present data from the model which runs externally on the target computer. This section tries to explain how RTW works, as a good understanding of the tool is need for the autonomous bike. Here, the goal is to explain the process of RTW, more so than to explain how to make it work. A more detailed description of how to start using RTW is given by [Fossum, 2006, Vedlegg 2]. New versions of Matlab has introduced some deviations from guidelines given by [Fossum, 2006, Vedlegg 2], and if so, this will be mentioned here.

2.3.1 Configuration

Figure 2.8 shows the RTW configuration pane. There are several important aspects, each treated separately. Target is the hardware or operating system where the generated code will run. Host computer is the computer running Simulink and RTW, and is responsible for file generation. The generation process is dictated by a *system target file*, *template makefile* and a *make* command.

The *make* command given here, *make_rtw*, tells Simulink how to start the build process.

The *system target file* is a `.tlc` file. This file contains specification of compiler command, build file and similar. TLC[13] is a compiler working in unison with RTW to create the final source code. TLC uses the model descriptions RTW creates using the Simulink model (clear text, `.rtw` files describing the model) and generates target specific code. Figure 2.8 shows a overview of the process.

The *template makefile*, a `.tmf` file, is a dynamic makefile containing variable fields. The fields are filled out based the Matlab and Simulink preferences, making it possible to customize the build process without editing the makefile. The template makefile has to contain the file paths for all libraries and external source code which should be linked to the build process. When embarking on the bike project it became apparent that the existing tmf file was incompatible with the newest (2009) Matlab version. Based on the general unix os `.tmf` files provided with Matlab, and guided by the previous bike `.tmf` file developed by [Lasse Bjermeland and Nessjøen, 2007] an new file was made. The file was named `qnx_unix_2009.tmf` as shown in Figure 2.8 .

[12] Real-Time Workshop

[13] Target Language Compiler

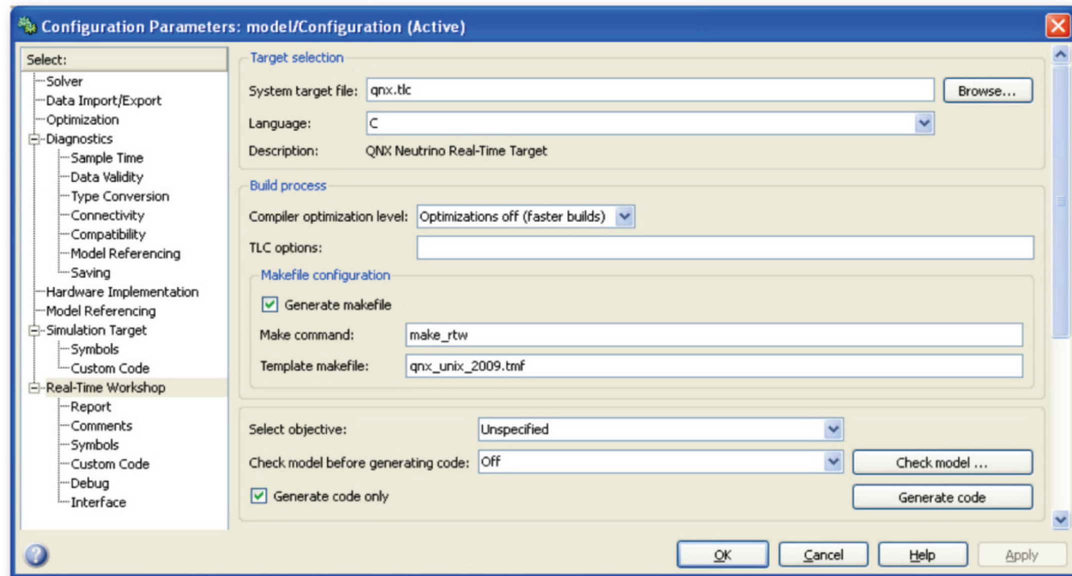

Figure 2.7: Configuration window for RTW.

In addition to the above mentioned files, a custom main function for the target computer is required to communicate with the host computer. For the QNX OS, such a file was developed by [Lasse Bjermeland and Nessjøen, 2007]. This file handles the IPC required for the TCP/IP protocol so Simulink can access data from code running on the bike.

2.3.2 S-functions

S-functions can be used as customized Simulink blocks which determines how the blocks are initialized, updated, outputting data and terminates. S-functions can work as an interface to different environments, hardware or software. For the bike, a S-function is written in C, similarly to how the rest of the drivers are done. The S-function contains two parts, one generated by Simulink and one custom written. `bike_io.c` is generated code for initialization, updating block state, outputting data and termination, serving as an interface to the custom written code. `bike_io_wrapper.c` is the custom written part, and the functions defined by the wrapper are called by the `bike_io.c` interface. The wrapper code is responsible for opening the resource managers running on the bike with *open()* during initialization and running *devctl()* for all managers every time the model is updated. When the model is stopped, the resource manager file paths are closed.

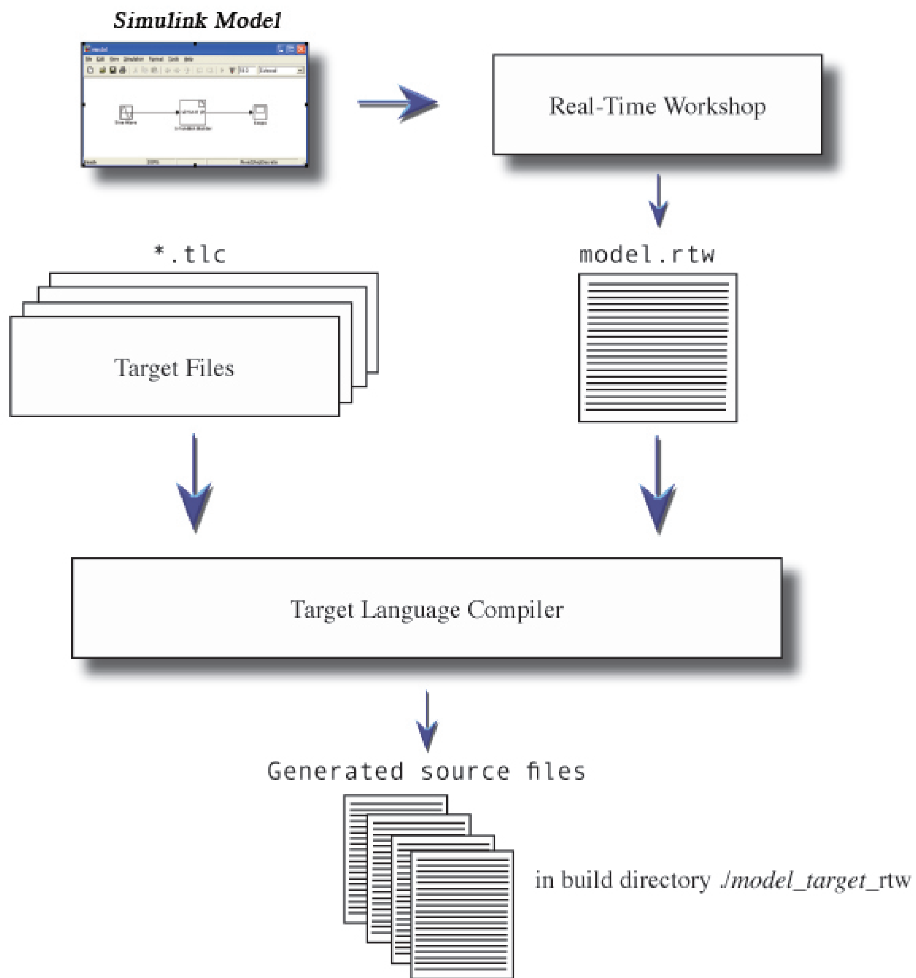

Figure 2.8: Broadly how RTW creates target files.

2.3.3 Communication

It is desirable for the host computer to be able to observe the model running on the bike through Simulink. This is achieved by assigning a IPC channel for the host and the client to use for communication. Figure 2.9 shows how the channel is selected, by setting *Interface* to *External*. Several targets are supported by Matlab "out of the box", unfortunately QNX is not one of them. With the introduction Matlab 2009, custom targets require a `sl_customization.m` file in the Matlab file path during startup. Code sample 2.4 shows the function used to allow TCP/IP communications for the bike. This file defines which IPC channels are available for the custom target, and selectable using the pull-down menu. When available *tcpip* is selectable under *Transport Layer*, and the IP-address for the target computer can be specified.

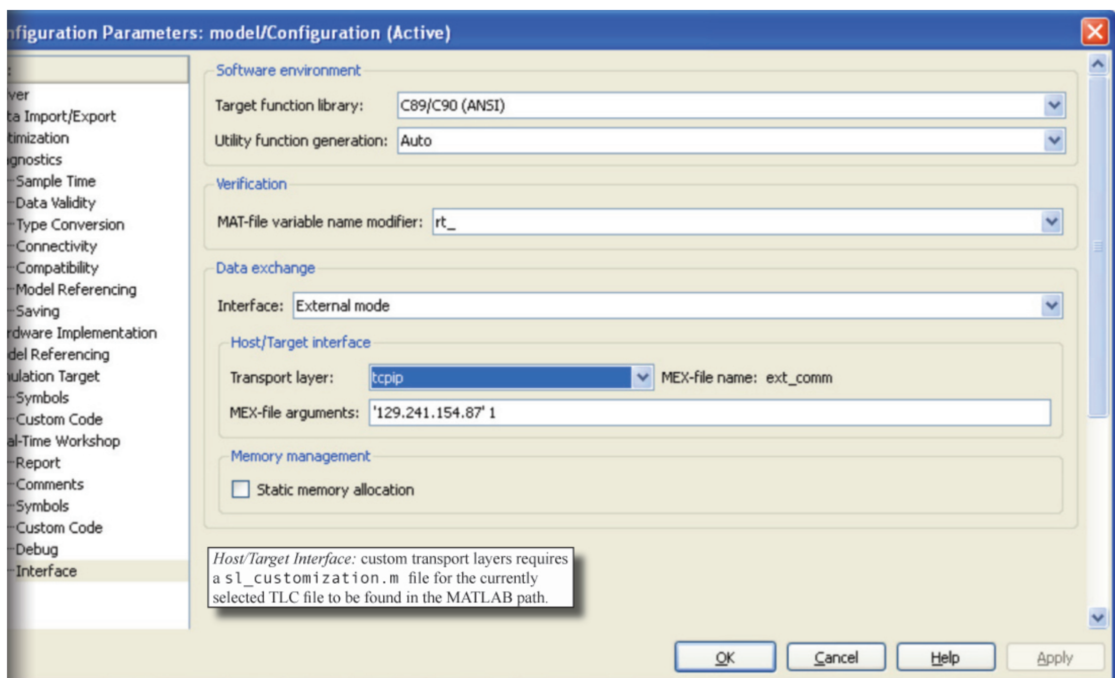

Figure 2.9: Interface settings for host/target communication.

Code sample 2.4 The `sl_customization.m` file, specifying available IPC-channels for a custom target.

```
function sl_customization(cm)
        cm.ExtModeTransports.add(,qnx.tlc,, ,tcpip,, ,ext\_comm,, ,Level1,);
end
```

2.3.4 Compiling and Running the Code

When the RTW has finished the build process, a makefile and source code has been generated necessary to compile an executable for the target machine. The compile process is dependant upon libraries specified in the *template makefile*, mentioned above. [Fossum, 2006, Vedlegg 2] gives a detailed description of the required Matlab folders, but note that the folders should be acquired from Matlab directly and not from earlier bike projects, for compatibility. The executable is compiled with the command

```
# make -f model.mk
```

model is the name of the Simulink-model. The executable can be run with

```
# ./model -tf x -w
```

x specifies the runtime of the model, in seconds. Alternatively, *inf* can be used to indicated infinite time, ie. the model runs until it is stopped. The *-w* flag indicates that the model should await a remote communication and start signal from the host running Simulink before starting.

2.4 System flow

Putting everything mention previously into context, the system flow of the bike can be explained. When the autonomous bicycle is started, the hardware sends data to the resource managers which in turn provides measurements and signals to the bike model via a S-function. The model updates itself based on the data provided in real-time, and transfers the model state over TCP/IP[14] to Simulink set to external mode on a host computer. The state of the bike is now observable from the host computer; a riderless bike has been set in motion.

[14]Transmission Control Protocol/Internet Protocol

Chapter 3

Hardware

Over the course of this chapter, a broad overview of the physical components found on the bike are given. It is meant as a point of reference for the various hardware available, and draws documentation from the master theis of [Sølvberg, 2007] and [Loftum, 2006], as well as various data sheets. The chapter describes the state of the bike at the end of this thesis, and Figure 3.1 shows the equipment currently connected to the bike. Peripherals such as network cables, keyboard, mouse, monitor and so forth are left out, but are also trivial to connect.

3.1 Current source

The current source used for DC/DC conversion is a ACE-890C from IEI Technology Corporation. The source is capable of delivering 18-36V DC, and requires 7A (RMS) at 24V DC. Refer to Table 3.1 for additional data, and Figure 3.2 where the ACE-890C is shown.

3.2 Computer

The autonomous bike is run by a Wafer-945GSE2 "Single-board Computer" (SBC[1]). The Intel Atom processor N270 embedded on the WAFER-945GSE2 has a 1.60 GHz clock speed, a 533 MHz FSB and a 512 KB L2 cache. The Wafer-945GSE2 also includes onboard 1.0 GB DDR2 SDRAM (as per IEI Technology Corp. [2009]). Table 3.2 gives an overview of some board specifications.

[1]Single-board computer

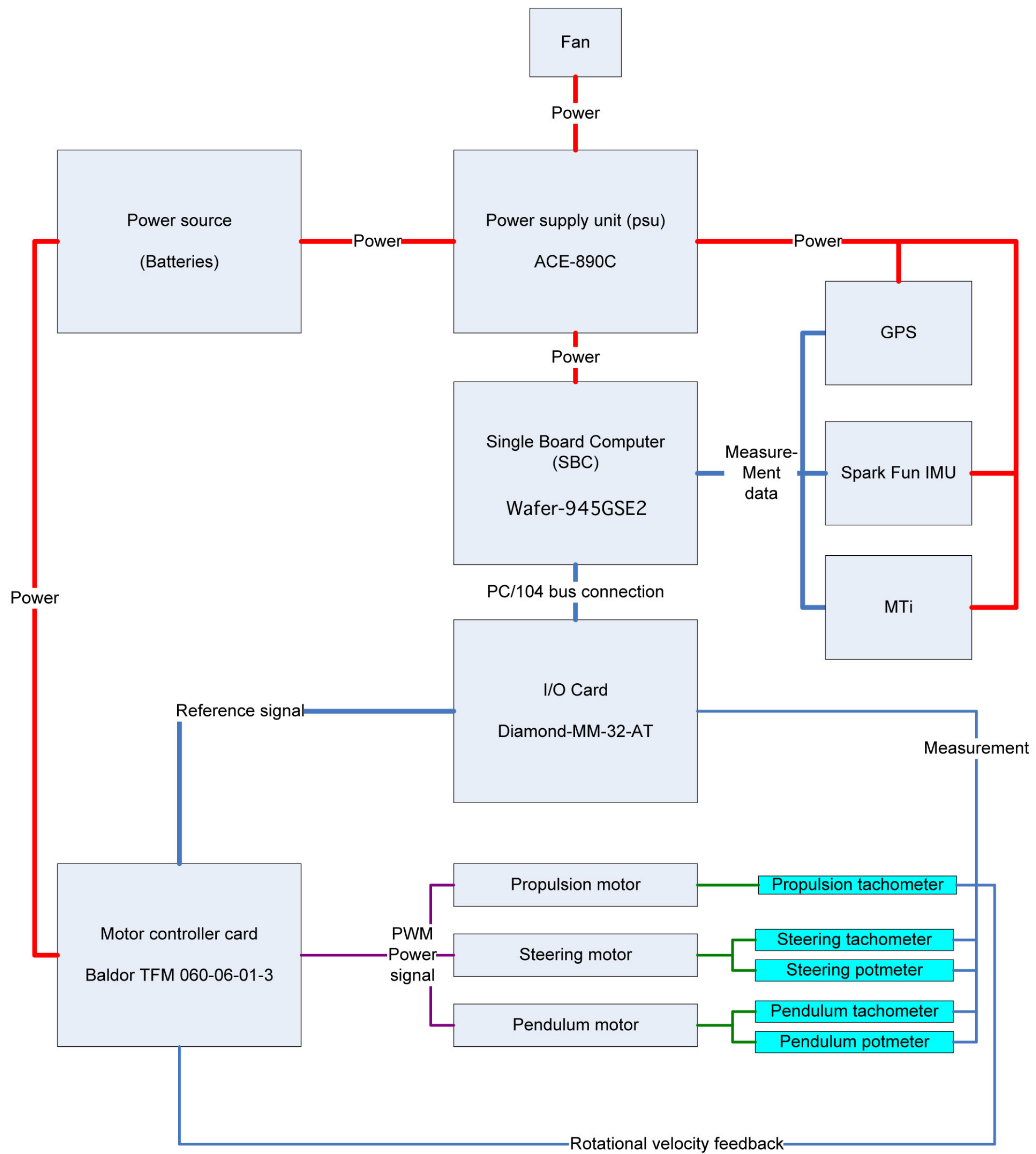

Figure 3.1: Overview of the current bike hardware. Based on picture from [Sølvberg, 2007]

Input	
Voltage	$18 \sim 36$VDC
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	$0 \sim 50\infty C(Operating)$
	$-20 \sim 85\infty C(Storage)$
Dimension	$152.4 \times 89 \times 39mm$

Table 3.1: ACE-890C specific data.

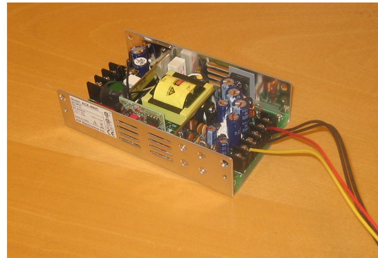

Figure 3.2: ACE-890C current source. Picture from [Loftum, 2006].

Figure 3.3: Wafer-945GSE2 SBC used by the bike. Picture from [IEI Technology Corp., 2010].

Parameter	Value
Product	WAFER-9371A
Form Factor	3.5" SBC
CPU	ULV Intel Atom N270 1.60 GHz
Memory	1.0 GB DDR2 SDRAM
Display	VGA 18-bit or 36-bit LVDS
	Dual display support
I/O Interface	1x 1.5 Gbps SATA 1 x Keyboard and mouse connector 1x RS-232/422/485 1x RS-232 1x PC/104 ISA 1x CompactFlash socket
Ethernet	2x 10/100/1000 Mbps RTL8111CP
USB	2x USB 2.0 connectors (supports 4x)
Audio	ALC655 AC'97
Power consumption	5V @ 3.1 A
WDT	1 $\sim$ 1-255 sec
Dimension	146 mm $\times$ 102 mm

Table 3.2: Specific Wafer-945GSE2 data.

Figure 3.4: Samsung HD322HJ 320 GB/7200RPM/16M hard drive. Picture from Samsung [2010].

3.3 Storage

The storage device is a Samsung HD322HJ with 320 GB of space operating at 7200RPM (shown in Figure 3.4). It is connected to the Wafer-945GSE2 with a SATA[2] cable, and receives power from a Molex connector.

3.4 I/O card

An I/O card is required to access the potmeter and tachometer measurements , as well as adjusting the voltage supplied to the motor controller card. The bike uses a Diamond-MM-32-AT 16-bit analog I/O modul, or DMM-32-AT, as show in Figure 3.5. The card is connected to theWafer-945GSE2 via a PC/104 connector.

Among the various functions provided by the card, the most relevant are mentioned here ([Diamond Systems Corporation, 2003]):

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[3] for reduced interrupt overhead
- Auto calibration of all input ranges under software control

[2]Serial Advanced Technology Attachment
[3]First In - First Out

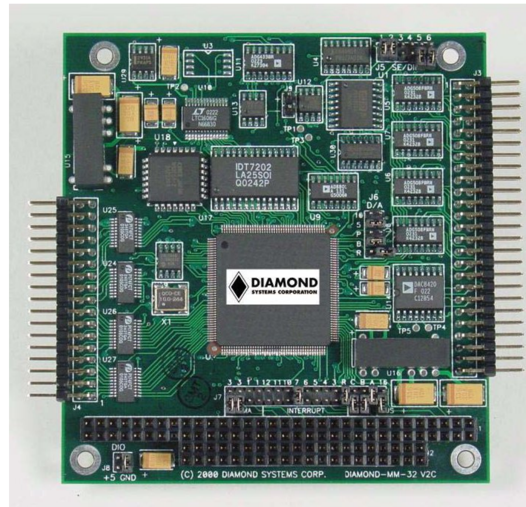

Figure 3.5: The DMM-32-ATcard, from [Diamond Systems Corporation, 2003]

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

Additionally the DMM-32-AT sports 24 bidirectional digital I/O lines, a 32 bit counter/timer for A/D[4] pacer clock and interrupt timing, and a 16 bit general purpose counter/timer, both with programmable input sources, and multiple-board synchronization capability. Section 4.3.1 describes the software driver responsible for controlling the I/O-card.

3.4.1 Motor controller card

The motors on the CyberBike are powered through a Baldor TFM 060-06-01-3, as shown in Figure 3.6. [Baldor ASR, 1988] provides data for the card, and some highlights are given here:

- 4 quadrant operation.
- 360 Watts possible continuous output power.
- 6A continuous phase current.

[4] Analog-to-Digital

- 12A peak phase current, for a short while.
- 24V nominal DC[5] 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 (≈ 14.7 % of the switching frequency of the pulse width modulated output signal).
- Ca. 80 % efficiency

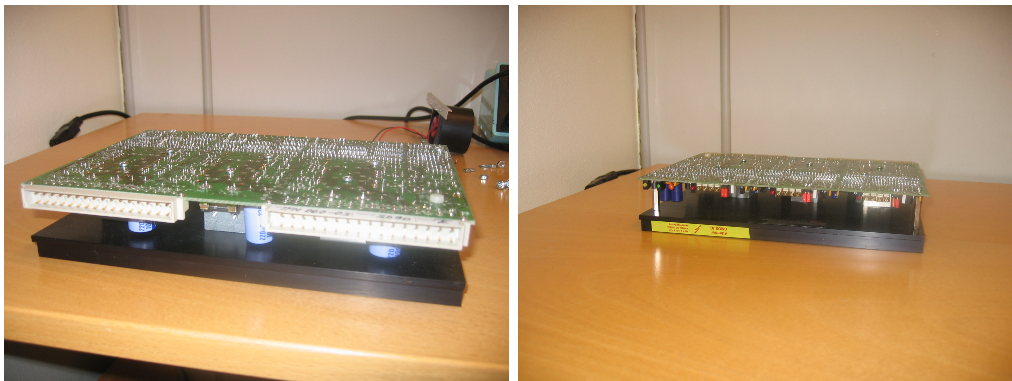

Figure 3.6: The CyberBike's motorcontroller card, a Baldor TFM 060-06-01-3. Seen both from the rear and the front. Pictures from [Loftum, 2006].

It has, during the course of this thesis, become clear that since the card is incapable of providing more than 6A continuous output current, which is insufficient for the propulsion motor.

3.5 Motors

The autonomous bike has three motors attached; one connected to the rear wheel, providing propulsion; one connected to the steer, providing directional steering; and one connected to the inverted pendulum, providing balance.

[5]Direct Current

3.5.1 Propulsion motor

The propulsion motor is made by [DtC-Lenze as, 2007] of the type 13.121.55.3.2.0, SSN31 and shown in Figure 3.7. [Sølvberg, 2007] gives a more throughout description of the individual numbers, and their meaning. Technical specifications are shown in Table 3.3. Note that the motor is rated for currents not supported by the motor controller card; this problem is discussed in Section 5.8.

	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 3.3: Specifications for the propulsion motor.

Figure 3.7: The propulsion motor; a Lenze Worm Geared motor. Picture from [Sølvberg, 2007]

3.5.2 Motor for the inverted pendulum

The pendulum motor is a ITT GR 63 x 55 TG11, operating at 24V DC and 4A nominal current. Nominal rotation speed is 3350 rpm. The motor is mounted a planetary gear

with ratio 79:1 and shown in Figure 3.8.

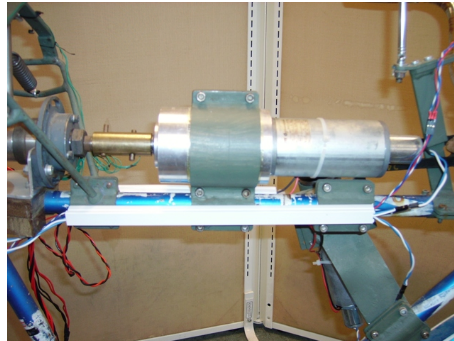

Figure 3.8: Pendulum motor; a ITT GR 63 x 55 TG11. Picture from [Fossum, 2006].

3.5.3 Steering motor

The steering motor is a ITT GR 53 x 58 TG11, operating at 24V DC and 4A nominal current. It is shown in Figure 3.9. Nominal rotation speed is 3000 rpm.

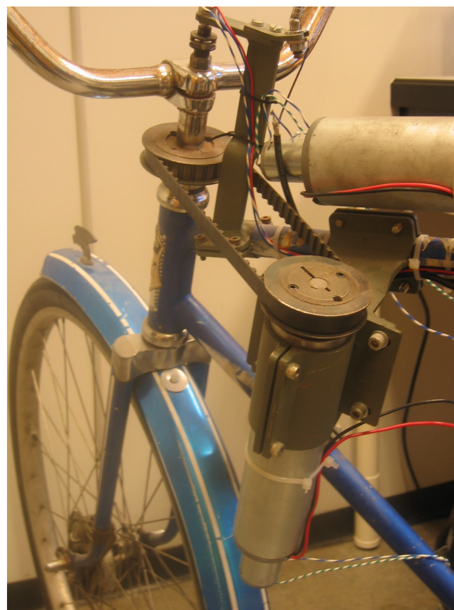

Figure 3.9: The steering motor; a ITT GR 53 x 58 TG11. Picture from [Loftum, 2006].

3.6 IMU - Xsens (MTi)

As the positional measurement device of choice, the IMU[6] from Xsnes dubbed "MTi" is available for the bike (shown in Figure 3.10). The MTi makes a 3D positional description through acceleration, turn rate, and earth magnetic field measurements. It is an excellent device for providing data to control scheme and system model, and provides Simulink access to the data through a S-function communicating with a resource manager.

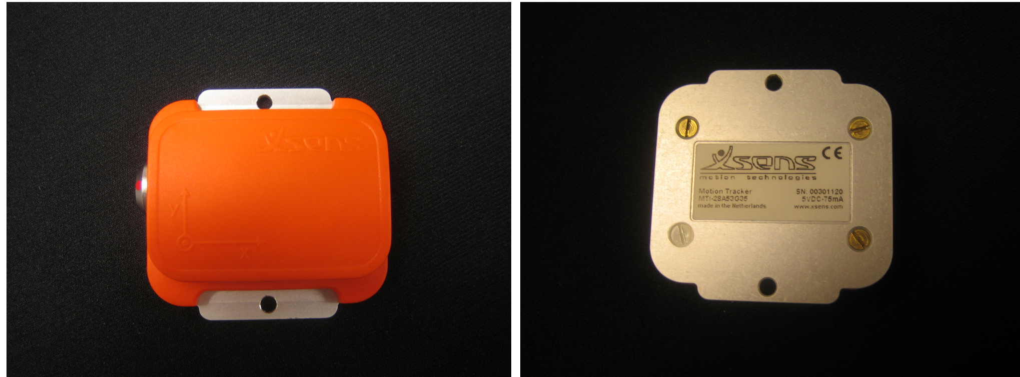

Figure 3.10: "MTi" from Xsens.

The MTi can send measurements over RS-232 or USB, but at this time, a RS-232 has been selected as the transfer method. The measurements are calculated relative to two reference systems. On is relative to the sensors, and the other relative to Earth, both as right-handed Cartesian coordinate systems where:

- X is positive when facing the magnetic north.
- Y is positive when facing west, relative to X.
- Z is positive along the vertical axis.

The data can be received as either rotational matrices, Euler angles or quaternions. The MTi driver handling the software communication with the device is covered in Section 4.3.3.

3.7 IMU - Spark Fun

The IMU from Spark Fun can at this point be considered an obsolete piece of equipment on the bike, as a better alternative is found in the MTi (Section 3.6). However, it is available and fully functional and can potentially be used given some additional work. Therefore a short description is provided here. The IMU consist of a small motherboard and three gyros as show in Figure 3.11. The card was originally installed to provide

[6] Inertial Measurement Unit

measurements from the gyros. The card has a PIC16F88 μC[7], integrated DAC[8] and a CD74HC4067 multiplexer. The multiplexer gathers the five measurement signals from the gyro cards and sends them to the μC. The data is then available via UART[9]. A description of the IMU driver can be found in Section 4.3.2.

Figure 3.11: IMU from Spark Fun Electronics with 6 degrees of freedom. Picture from [Loftum, 2006]

3.8 GPS

The absolute position of the bike can be determined using the GPS-module available on the bike. It is a *GlobalSat EM-411* and shown in Figure 3.12. Sølvberg developed and installed a PCB card, to allow the Wafer-945GSE2 to access the GPS-module (show in Figure 3.13). A description of the GPS-driver interfacing with the device is given in Section 4.3.4. At this point the GPS-module has not been used in any actual application on the bike.

3.9 Emergency Stop

The bike has an emergency stop button accessible from the outside of the cabinet housing, to increase the safety of the system. Figure 3.14 shows the button. Generally the emergency stop should be activated during startup of the bike, as a standard safety precaution.

[7] micro controller
[8] Digital-to-Analog Converter
[9] Universal Asynchronous Receiver Transmitter

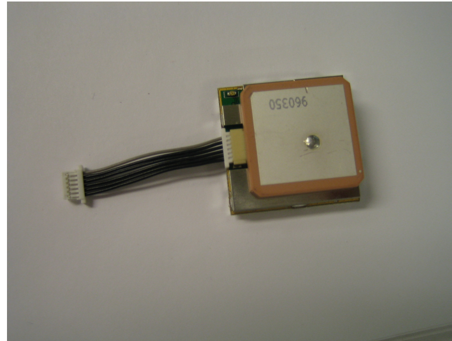

Figure 3.12: EM-411 GPS from GlobalSat.

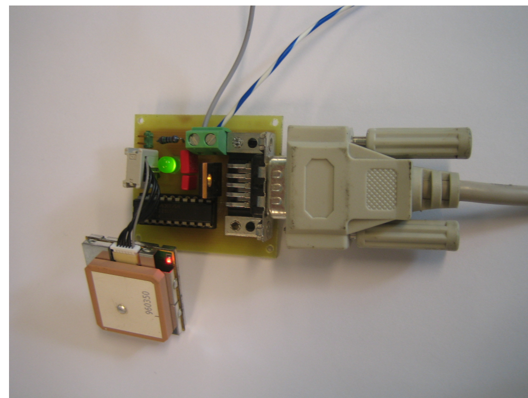

Figure 3.13: Component side of the GPS-card, showing the PCB.

Figure 3.14: Emergency Stop. Picture from [Sølvberg, 2007].

3.10 Fan

To avoid overheating the electronics contained in the cabinet, a 12V fan is installed in the cabinet wall. This is connected to a switch, only enabling the fan whenever the cabinet cover is closed. This has been done to avoid unnecessary noise during development.

3.11 Batteries

Two ATU12-35 batteries from [ACT Batteries, 2007] are available to power the bike on demand. They provide 12V each, and has a total weight of 21 kg, considerably increasing the weight of the autonomous bike. Each battery is mounted in a separate frame, distributing their weight equally on both sides of the bike. A power switch accessible on the outside of the cabinet can toggle the system on or off.

3.12 Pendulum Limit Switches

The inverted pendulum can inflict severe equipment damage if allowed to spin out of control for whatever reason. To combat the problemSølvberg installed limit switches which triggers whenever the pendulum reaches a certain angle in either direction. The intent was to prevent the motor from applying pressure when the pendulum reaches its end points, thereby preventing damage to the motor brushes. Figure 3.15 shows how this has been implemented.

The current implementation stops all power to the motor when a switch is activated, preventing the pendulum from being controlled in either direction. The implementation will not prevent the pendulum from crashing with the frame, as gravity and current inertia will still take its course - there is no velocity control.

During the course of the thesis, the limit switches where shown to be mounted in such a way that they severely restricted any control scheme from functioning properly. As such, they where unscrewed to allow for greater pendulum movement. However, it is a simple task to reattach them to their original position, should this be desirable.

3.13 Potmeters

The bike sports two potmeters, used to measure the angle of the steer and the pendulum. Figure 3.16 shows the potmeter connected to the steer. The potmeters vary from 0 to 5V based upon the angle. After the signal has been received by the Diamond-MM-32-AT, the signal is interpreted as a value from -5V to 5V.

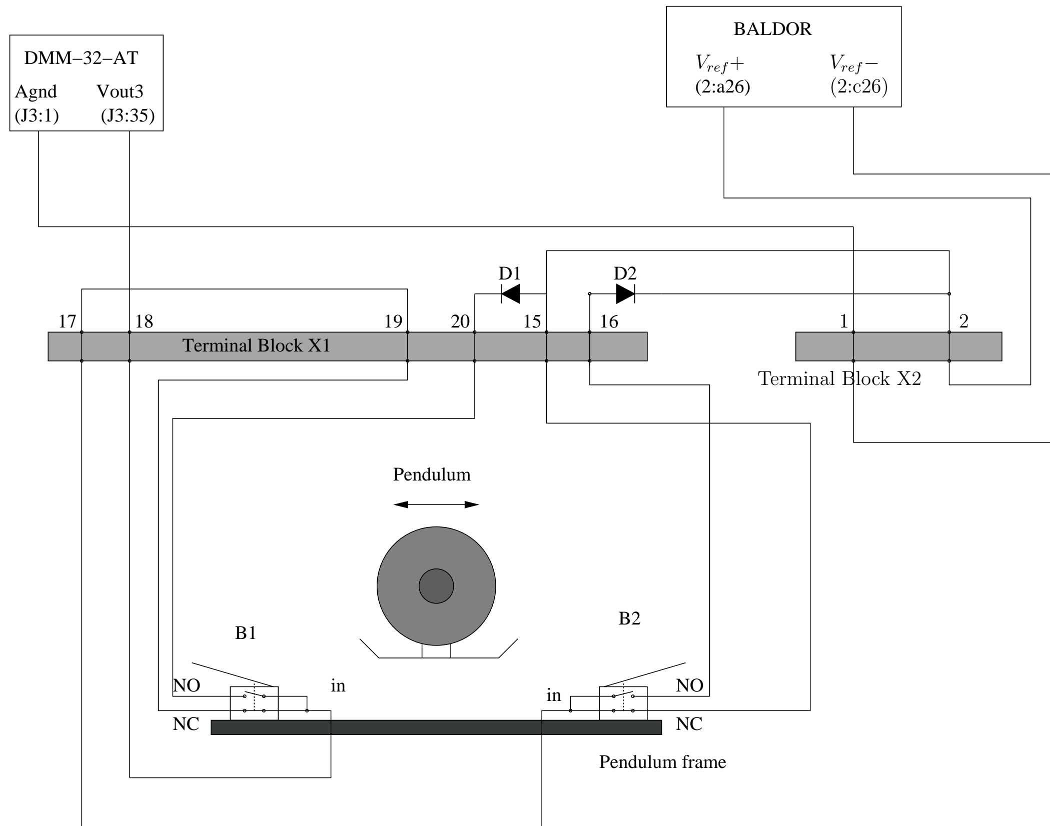

Figure 3.15: Pendulum limit switch circuit. Figure from Sølvberg.

Figure 3.16: Steer potmeter.

Chapter 4

Software

4.1 Software

This chapter aim so cover the software available on the bike computer. The following sections will cover the use, and briefly, the implementation of the software, based on the work done during the course of thesis, and the work done by [Sølvberg, 2007] and [Loftum, 2006]. In the same way that Chapter 3 is meant as an overview and starting point for future development, this chapter gives a description of the work done thus far. For a more comprehensive and throughout explanation of the various drivers, references to relevant master thesis are given where needed.

4.2 OS[1]

The bike runs QNX Neutrion RTOS 6.4.1. This is a fully fledged RTOS, displayed through the Photon graphical user interface. The OS was chosen because the autonomous bike requires strict real-time response. QNX is developed with embedded systems in mind, and has been widely used for exactly systems such as this. Section 2.2 covers many important concepts related to QNX, and how they relate to the bicycle project.

4.3 Drivers

The following sections gives a description of the bike drivers, focusing on their *use*. The drivers are all implemented as resource managers and adheres to the principles described in Section 2.2.2. The drivers has to be started separately after QNX has booted, optionally with the *&* flag added to the start command. This makes the driver run in the background.

Typically the resource managers contain a string array with file paths to device files.The drivers are accessed through these files. During initialization of a driver the

[1] Operating System

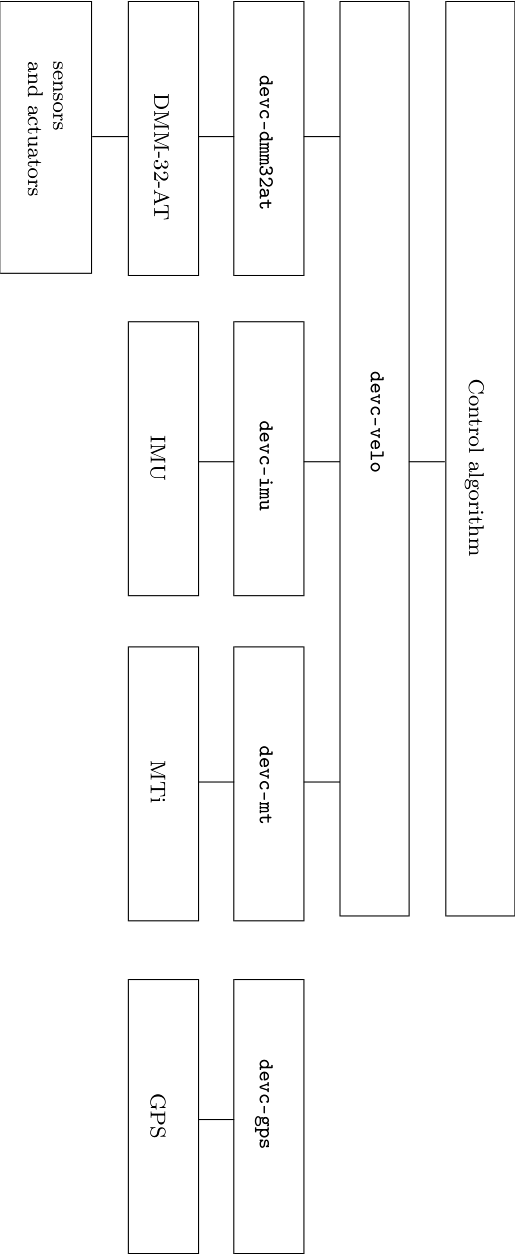

Figure 4.1: Overview of the modular software structure of the bike.

file paths are registered with the OS, as well as a set of *open()*, *read()*, *write()* and *devctl()* function calls for each device file. The subsections indicate where the resource managers put the device files and their file paths for easy reference, as well as any optional modifiers available for the driver. Figure 4.1 show how the drivers are separated into different modules, and which parts are dependant upon each other. The drivers should be started from the bottom up, in other words, `devc-velo` should be started last.

During the thesis, the ability to run the drivers without having the actual hardware connected was implemented. If a driver fails to detect hardware, it will be running in "zero-return mode". This basically means that the driver will populate the file path with device fils which will return 0 or the equivalent whenever they are read. This allows the driver to be run on any QNX system, making it possible to debug, for instance, the Simulink models by connecting to the QNX development PC instead of the bike PC. This has been very useful thorough the working process.

4.3.1 Resource Manager: `devc-dmm32at`

File paths: `/dev/dmm32at/analog/out`, `/dev/dmm32at/analog/in`

Optional modifiers: [-v] [verbose] [autocal]

The DMM-32-AT-driver was initially written by [Loftum, 2006], future developed by [Sølvberg, 2007] and finally modified during the course of this thesis. The driver initializes and controls the I/O card on the bicycle and handles A/D-conversion from analog in channels and D/A-conversion to the analog out channels. A resource manager thread process request from clients. *write()* calls made on the device files in the `out` file path, converts the argument value given to a 16-bit number which is forwarded to the D/A-converter. The signal is used to control the voltage to the motors, and can be done using a terminal window as described in Section 2.2.2. The driver is started with the command:

```
# devc-dmm32at [-v]
```

The *-v* flag is optional. If provided, the flag will put the driver in verbose mode, making the driver output debug information. Alternatively, *verbose* can be used, and serves the same purpose.

Client programs can communicate with the driver by using *devctl()* call. Code sample 4.3.1 shows an exaple of this. The variables *fd1* and *fd1* are file descriptions referencing the device files in question, and the *val* is the value to be written or read. *DMM32AT_DEVCTL_GETVAL* and *DMM32AT_DEVCTL_SETVAL* are macros which defines the direction of the data transfer. This implementation is very similar for all the device drivers.

[Sølvberg, 2007] discovered that DMM-32-AT has to be calibrated now and again. This should only be performed when da0 is disconnected from the J3-header on DMM-32-AT. In effect, this means that much of the connector to the I/O card has to dislodged

Code sample 4.1 Example: Usage of *devctl()* with `devc-dmm32at`

```
int fd1, fd2;
double val=2,5;
if ((fd1=open("/dev/dmm32at/analog/in/ad1", O_RDWR)) == −1) {
    printf("Couldn,t open file!\n");
    return(1);
}
if ((fd2=open("/dev/dmm32at/analog/out/da2", O_RDWR)) == −1) {
    printf("Couldn,t open file!\n");
    return(1);
}

ret = devctl(fd1, DMM32AT\_DEVCTL\_GETVAL, &val, sizeof(val), NULL);
ret = devctl(fd2, DMM32AT\_DEVCTL\_SETVAL, &val, sizeof(val), NULL);
```

.

before auto-calibrating, as just disconnecting the da0 channel is impossible. Calibration can be initiated using the following command:

```
# devc-dmm32at -v autocal
```

The -v flag is optional, but recommended. When the calibration is completed, the driver exits. Be sure to use this feature if no hardware fault can be found in connection with signal drift from the potmeters.

This driver is responsible for applying power to the motors. The device file da0 controls the propulsion motor, d1 the steer motor and da3 the pendulum motor. A simple way to test a motor can be done as following:

```
# echo 0 > /dev/dmm32at/analog/out/da1
# echo 1 > /dev/dmm32at/analog/out/da1
```

It is advisable to always start testing by echo 0 to the device file, so the motor can be stopped quickly through the terminal history.

4.3.2 Resource Manager: `devc-imu`

Main file path: `/dev/imu`

Sub paths: `/pitch/`, `/roll/`, `/yaw/`, `/battvoltage/`

Optional modifiers: [-v] [verbose]

The IMU driver is a resource manager written by [Loftum, 2006] and future developed by [Sølvberg, 2007]. The driver has one thread reading data from the serial port and one resource manager thread intercepting client requests. the IMU starts transferring data whenever a ASCII '7' sign is received on the serial port. 34 byte packets are sent at 23.5Hz in a specific pattern.

The stream starts with an ASCII 'A' sign, proceeded by 16 DAC measurement values in the following order:

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

Finally the stream completes by the ASCII 'Z' sign, simplifying synchronization. The measurements are available through the device files, one for each measurement.

The driver can be started with the following command:

```
# devc-imu [-v]
```

The *-v* flag is optional and puts the driver in verbose mode. The flag will make the driver output debug text to screen during run time. Note that the IMU driver is hard-coded to use COM port 2 to interface with the IMU device.

At the time of writing, this driver can be considered obsolete in the sense that the MTi can do everything the IMU can, but better. Furthermore, the data provided by the IMU cannot be used by Simulink in a trivial manner, and as such renders the device unpractical.

4.3.3 Resource Manager: `devc-mt`

Main file path: `/dev/mt`

Sub paths: `/calib/acc/`, `/calib/gyr/`, `/calib/mag/`, `/orientation/`, `/samplecounter/`

Optional modifiers: [-v] [-vv] [-m3] [-f3] [-s /dev/ser2]

Important note: At the time of writing, this particular driver does not support the [] modifier. Ie. it cannot be run in the background. This is caused by some a stack overflow on the serial device, which results in the driver terminating if it is put in the background. Therefor the driver has to be run in its own, seperate terminal window.

The MTi-driver is a recourse manager developed by [Sølvberg, 2007], based on the standard driver available from Xsens. The driver, similarly to the IMU-driver, runs two thread. One thread read data from the serial port, and one thread serves the role of the resource manager, managing client requests. The measurements from the serial port are stored in a measurement array, protected by a mutex. There is currently no *write()* function support, as only data acquisition is required from the gyrometer.

The driver can be started by typing:

```
# devc-mt -vv
```

This starts the driver in noisy mode, showing calibration and orientation data continuously in the terminal. Printout 4.3.3 shows the use-file available for the driver, it explains the driver options in more detail.

```
Syntax:
devc-mt [options*]

Options:
-m [mode] Output mode. Available modes are:
                    1 - Calibrated data
                    2 - Orientation data
                    3 - Both Calibrated and Orientation data (default)
-f [format]       Output format. Available formats are:
                    1 - Quaternions
                    2 - Euler angles (default)
                    3 - Matrix
-v Be verbose ( -vv will give noisy mode)
-s [com_port] Specifies which device file name to read from (default /dev/ser1)

Examples:
       devc-mt -m 3 -f 1 -vv -s /dev/ser2 &
```

4.3.4 Resource Manager: `devc-gps`

File path: `/dev/gps/`

Optional modifiers: [-v] [-vv] [verbose] [noisy]

The GPS driver is a resource manager written by [Sølvberg, 2007]. It has one thread responsible for acquiring data from the serial port (hard coded to /dev/ser2), and one resource manager thread responding to client requests. Received gps messages are analyzed by `gps-messages.c` which is run by the serial port thread. There is one function of every type of message. The functions store the values to local variables, locks a mutex, store the measurements in an array, before unlocking the mutex. [Sølvberg, 2007] goes into greater detail about how the measurements are stored and used throughout the code. Sufficient to know to use the drivers is that there is one device file for every measurement in the array, all available through the `/dev/gps/` file path.

4.3.5 Resource Manager: `devc-velo`

File path: `/dev/velo/`

Optional modifiers: [-v] [-vv] [-a number] [-q net_path]

This driver is the top level interface to all the bike measurements and motors. The driver is ultimately responsible for communicating with the bike model through an S-function. It was originally developed by [Loftum, 2006], future modified by [Sølvberg, 2007] and reaching its current state during the course of this thesis. `devc-velo` depends on the other drivers presented thus far to populate its device files. Whenever this resource manager is accessed, it uses *devctl()* calls to the other drivers, scaling the data as necessary before finally replying to the client request. As such, `devc-velo` cannot be started before the other drivers are running.

`devc-velo` can be started with the following command:

```
# devc-velo [-v]
```

The *-v* flag is optional and indicates to the driver that it should run using verbose mode. Printout 4.3.5 shows the use-file available for the driver. By default the IMU driver is not sampled, but it is possible to select either or both of the gyrometers. The *-q* flag makes it possible to specify a remote path for communicating with the MTi resource manager. This makes it possible to have the MTi device connected to a remote computer, but still have the bike receive orientation data. Section 5.5 discusses why this was implemented.

```
Syntax:
devc-velo [options*] &

Options:
-q [net_path] If specified, the driver will remotely aquire MTi data. Disabled by default.
                     Example:
                            "-q /net/dev_pc" will set the path to the MTi device driver to:
                            /net/dev_pc/dev/mt
-v Be verbose ( -vv will give noisy mode)
-a [number] Selects the preferred accelerometer.
                     Available options are:
                            1 - Both
                            2 - MTi (default)
                            3 - IMU

Examples:
       devc-velo -q /net/dev_pc -a 1 -vv &
```

A preferences file, `potmeter.cfg`, located under `/root`, is used by the driver to adjust the minimum and maximum voltages for the steer and pendulum potmeters. Bias values representing the offset angle required for the angle measurement to return zero when the steer and pendulum is centered is also set through this file. This simplifies the tuning process of the potmeters, should they have to be readjusted.

Refer to Section 2.3.2 for an explanation of how the driver can be used to communicate with Simulink by the use of a S-function.

4.3.6 Simulink Bike Demo

A Simulink model, capable of controlling the steer and the pendulum by using the MTi roll and pitch values as control variables is available. It is intended for a simple way to steer the bicycle as a "proof of concept" implementation. The controller gains are tunable through Simulink by adjusting tunable gain blocks, found in the controller subsystems. Figure 4.2 shows the overall model.

For the control scheme to work, the model has to be build by RTW and compiled for QNX as described in Section 2.3.4. Before starting the executable on the bike PC,

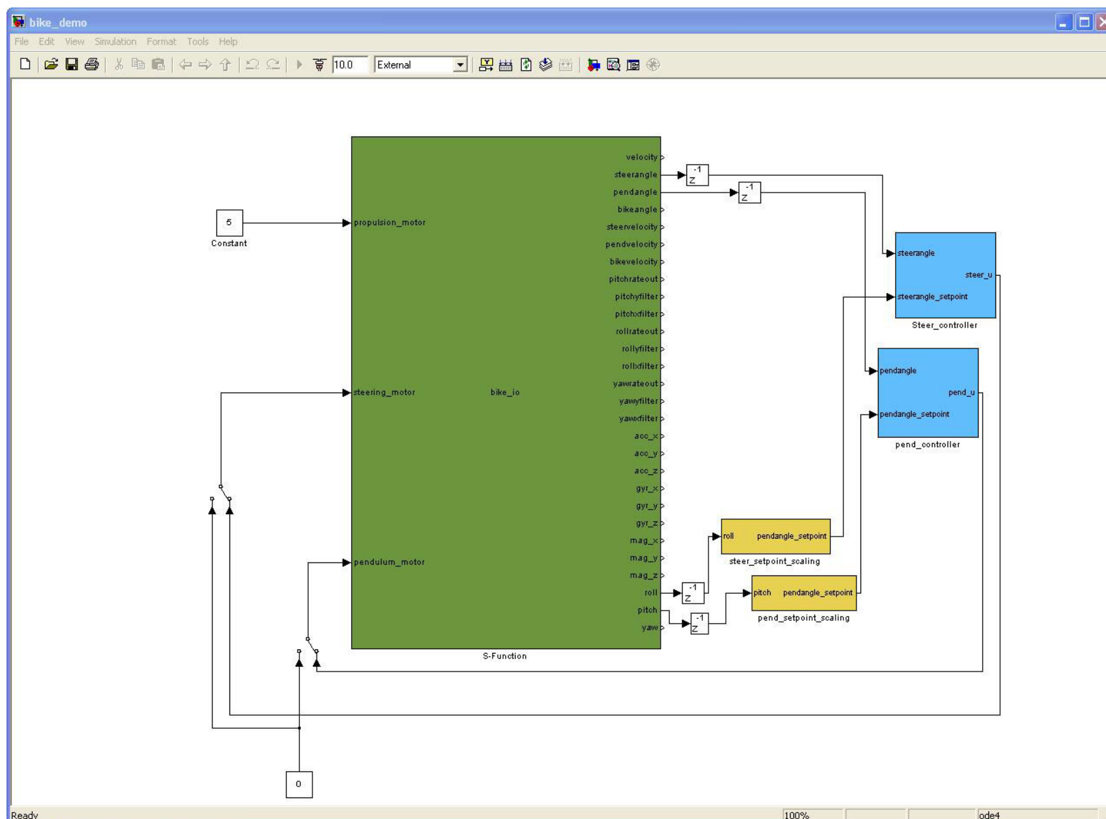

Figure 4.2: The bike demo model.

at the minimum, `devc-dmm32at`, `devc-mt` and `devc-velo` should be running in default mode (no modifier argument). The executable can than be started in wait mode (see Section 2.3.4), and the host computer can connect using external mode to the bike PC. The bike IP can be obtained by using:

```
# ifconfig
```

After the host has connected, the model can be started, and the bike controlled using the MTi device. For anyone new to the system, a step by step guide of how to test the Bike Demo with the development environment used during this thesis, please refer to Chapter B.

Chapter 5

Problems and Solution

During the the fall project preceding this master thesis, several issues affecting the autonomous bike became apparent. The thesis work itself gave rise to new problems as well, some of which where solved and some of which where left unfixed.This chapter will cover the obstacles and hurdles laying in the way of a complete control system, the actions taken to rectify the problems and the issues that remain. Problems where fixed on a priority basis, where the most pressing issues being handled first. Several minor issues where allowed to remain in the system, if workarounds where possible. The issues still plaguing the system, serve as points of discussion in Chapter 7.

5.1 New Motherboard

[Sølvberg, 2007] had the misfortune of working with broken COM ports on the single board computer that previously operated the bicycle. To allow communication with the COM peripherals like the MTi and GPS, the ports had to be repaired, or the motherboard replaced. Repairing the ports on the existing motherboard had the benefit of limiting monetary costs, but have the price of an unpredictable time frame. Additionally, [Sølvberg, 2007, Chap. 6.6] suggested that the 400 MHz of the Wafer-9371A could be a limiting factor in the control system, preventing the Simulink model from executing more than a few time steps. As such, the decision to upgrade the motherboard was made, and the Wafer-945GSE2 was purchased.

Some of the differences between the Wafer-9371A and the Wafer-945GSE2 are shown in in Table 5.1. As can be seen, the new board is significantly more powerful than the old, and can have more peripherals attached, at a modest increase in power consumption. The motors already require significant current to operate, so 1A increase is quite small for the overall system.

The new motherboard has only SATA connectors available for accessing external hard drives. As the existing Fujitsu MHK2060AT HD was an EIDE[1] drive, it could

[1]Enhanced IDE

Feature	9371A	945GSE2
CPU	400 MHz	1.6 GHz
RAM	256 MB	1.0 GB DDR2
USB ports	2x USB 1.1	4x USB 2.0
Ethernet	1x 10/100 Mbps	2x 10/100/1000 Mbps
Power consumption	5V @ 2.01A	5V @ 3.1A

Table 5.1: A comparison between Wafer 9371A and 945GSE2 .

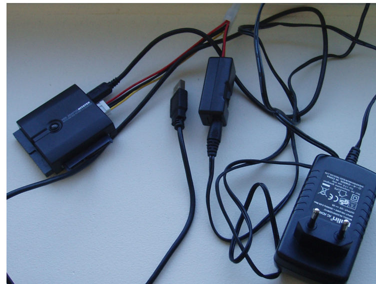

Figure 5.1: Hiyatek SATA/IDE HDD to USB 2.0 Adapter.

not be accessed using an EIDE cable directly. However, as the Wafer-945GSE2 sports two immediately available USB[2] connectors, an IDE to USB converter could serve as a temporary solution to the problem. After unmounting the motor controller card hiding the hard drive in the bike cabinet, the Fujitsu HD was connected to Wafer-945GSE2 via a Hiyatek SATA/IDE HDD to USB 2.0 Adapter (shown in Figure 5.1). Accessing the hard drive in this fashion is expected to preform poorly, as it introduced unwanted overhead compared to using the drive directly. Only the most basic peripherals where connected in addition to power and hard drive; a monitor, mouse, keyboard and network cable.

When applying power, the system boot sequence completed successfully, and presented the Photon login GUI. As expected, the startup time was longer than it had been on the previous motherboard, due to the USB converter. However, the screen resolution was very low, a maximum of 640x480, and a change in the Wafer-945GSE2 BIOS[3] was necessary to increase it. After the change had been made and the system rebooted, the screen could be set to a more comfortable size.

At this point, the autonomous bike had QNX 6.3.0 SP3 installed. As it turned out, 6.3.0 does not support the Realtek RTL8111CP Ethernet controllers on the Wafer-945GSE2 . Therefor, no networking was possible at this time. However, an OS upgrade was already planned, and under 6.4.1 RTL8111CP was known to be supported.

Later, when the hard drive, OS and network issues had been solved, a dual USB cable was connected to Wafer-945GSE2. The bike computer is therefore capable of having 4

[2]Universal Serial Bus
[3]Basic Input/Output System

USB devices connected at any one time. Unfortunately, the dimensions of the dual cable where different from dual cable used by the previous motherboard, preventing it from being mounted onto the bike cabinet. At the completion of this thesis, a firm mounting of the cable is left undone, but it is fully functional and available for use.

Section 6.2 gives a description of how the Wafer-945GSE2 was tested.

5.2 New Harddrive

As a direct consequence of the new motherboard, a new hard drive had to be acquired. The USB converter used for accessing the old drive could only serve as a temporary workaround, as it made the system slower, more complex and required additional power. Furthermore, as the final nail on the coffin for the Fujitsu HD, the hard drive stopped working shortly after QNX 6.4.1 was installed. No exact testing was done to determine the cause, but the hard drive, originally a relic from the Eurobot project, was believed to simply have reached the end of its lifespan. As not to delay the work on the thesis a new Samsung HD322HJ 320 GB/7200RPM/16M hard drive was bought. This is a cheap, yet new SATA harddrive. Optimally, the bike should have a flash drive with no moving parts, as a HD with mechanically moving parts is prone to damage if the bike should run out of control.

At this point it was necessary to install QNX Neutrino RTOS 6.4.1 onto the new hard drive. The QNX stationary host workstation did not have SATA ports, so the USB converter was used to connect the new drive to the workstation. A DVD with the OS was used to install QNX 6.4.1.

The disk was plugged into the QNX stationary host workstation, and the OS installed per the guidelines in [QSSL, 2005, chap.3]. The GNU Public License where chosen to be included in the installation. Using the workstation to install the OS is possible because both the stationary and the bike CPU are X86 Intel based computers. When prompted for a reboot, the system was completely shut down, the converter unplugged, and the hard drive connected to the Wafer-945GSE2 directly using a SATA cable. When the bike system was powered up, it booted significantly faster than before, but upon reaching the QNX login screen, the screen froze, exhibiting signs of a system crash. Some testing and research revealed (Section 6.2.4) that the culprit was having DMA[4] enabled.

Under QNX, DMA is enabled when the `.boot` file is loaded during startup. It can be disabled by pressing 'D', or by pressing 'space' when prompted. Pressing space will load the `.altboot` file instead of `.boot`. The files are identical in every way, except for the fact that `.altboot` has DMA disabled. It is unreasonable to expect a system to require user input during startup for it not to crash. To circumvent the problem, the `.boot` and `.altboot` files where swapped, in effect making `.altboot` the default boot file. This was done by making `qnxbase.ifs` the .boot file and `qnxbasedma.ifs` the .altboot.

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

[4]Direct Memory Access

5.3 OS Update

It is desirable to have the autonomous bike running software that is as up to date as possible, without affecting system stability. When starting the thesis, QNX 6.3.0 was an outdated OS, with the newer 6.4.1 available. Research also revealed that 6.3.0 has no support for wireless networking, so based on that fact alone a system upgrade seemed prudent (QSSL [2009]). A public beta version of 6.5.0 was available (in fact, the release version became available towards the end of the thesis), but running beta software can lead to unknown pitfalls by its very nature, so 6.4.1 was chosen as the new OS version.

The OS was installed on the development workstation as well as the bike PC. A minor problem with the introduction of QNX 6.4.1, is that the QNX Momentics Development Suite is no longer available for QNX itself, so all code development using the development suite has to be done using a Windows computer. As such the MDS was installed on the Windows workstation.

QNX 6.4.1, with support for the Wafer-945GSE2 Ethernet ports, made networking available without requiring additional setup. The bike PC was given the hostname "bike", while the development PC was given the name "dev" using the following command:

```
# hostname name
```

Here, "name", is the name by which the different computers will be identified over Qnet. Qnet was enabled on both computers with:

```
# touch /etc/system/config/useqnet
```

This creates an empty file in the indicated directory, indicating to the system that qnet should be enabled during startup. Reboot was required for qnet to start, after which the development workstation and the bike PC appeared under `/net` as `dev` and `bike`, respectively.

An attempt was made to compile the bike drivers under the new OS version, but this failed. Poking around revealed that the `sys/neutrino.h` apparently is buggy under 6.4.1, as it is missing a code piece. Some tidbits of code had to be added to the file to rectify the problem, as shown in Code sample 5.1. This allowed the bike drivers to compile successfully.

Another minor issue plaguing the compile process of the generated Matlab files from the Simulink build process where "//" comments. "//" comments seemed to be unsupported under the new OS, as they produced all sorts of wierd compile errors. The problem was fixed by replacing all "//" comments with "/* */" comments. However, as the files had compiled without problems under 6.3.0 this seemed a little odd. No in depth testing was done to find the cause of the problem, as adhering to ANSI C coding standards solves the problem.

Code sample 5.1 Missing code in `sys/neutrino.h`.

```
#if defined(__CLOCKADJUST)
struct _clockadjust __CLOCKADJUST;
#undef __CLOCKADJUST
#endif

#if defined(__ITIMER)
struct _itimer __ITIMER;
#undef __ITIMER
#endif
```

What immediately springs to mind, however, is the .tmf file. The target make file might not correctly identify 6.4.1 as a QNX system, and as such is using different compile flags in the final generated Makefile. The compile errors are similar to what would happen when using `-ansi` instead of `-std=c99`.

The bike driver executables where put under `usr/bin`, so they can be started by name anywhere in the system (through the terminal).

5.4 The Cyberbike Model

The Cyberbike Simulink model was developed by Bjermeland [2006]. Cyberbike is here referring to the actual Simulink model (not to be confused with Sølvberg [2007] name of the autonomous bicycle project). Sølvberg [2007] did some work to connect the model to the bike drivers, but was unable to complete it when the COM ports of the old motherboard stopped working. A lot of work was put into finalizing this model and to make it work as intended. Unfortunately, several issues plague the model, tied to the physical components of the bike and the nature of the bike drivers. Section 6.5.1 covers how the model was tested.

Sølvberg [2007, Chap. 6.6] failed to make the model run more than a few steps before stopping. He speculated that the processing power of the current CPU might have been exhausted, due to the complexity of the model.

It was determined that the Cyberbike model cannot operate at a sample rate quicker than 0.15 seconds. Verbose mode cannot be enabled for any of the drivers, as *printf()* introduces significant delay to the driver response. These restrictions will most likely apply to any use of the bike drivers. However, if not all device files needs to be sampled, the sample time can probably be improved.

The `bike_io_wrapper` code developed by Sølvberg [2007] did not work as intended, as the the device files was incorrectly ordered. This was fixed, along with adding some additional safeguards during the start, update and stop functions. The code was also using fd[0] instead of fd[i] when setting the output for the motors, in effect only controlling the propulsion motor. Along with this, some of the for loops where using "$<$" signs

instead of "<=". As Sølvberg [2007, Chap. 6.6] himself noted; the code was largely untested.

Whenever the estimator was used for the model, the system tried to apply exponentially increasing power to the motors. Disabling the estimator and forwarding the measurements directly avoids the exponential behaviour, but the pendulum and steer oscillates continuously, showing no sign of stability. The main culprit is believed to be the way torque is modeled in the bike drivers. Currently torque is reported as the voltage applied to a motor multiplied by 10. An alternative to this has not been explored, but is a problem that should be solved for a safe and stable system.

Working past many of the issues that presented themselves when testing the Cyberbike system, will most likely require a major overhaul of the bike model. As the focus of this thesis was not on the theory behind the bike model implementation, this has not been contemplated at all. However, the Cyberbike subsystem, serving as the interface to the bike measurements has been brought as up to date as possible, hopefully making it easy to connect to an enhanced model tackling these issues. The subsystem routes the measurements to the plotting facilities made by Bjermeland [2006]; the 3D model of the bike now uses to the physical position of the steer and pendulum, as well as the MTi accelerometer values for orientation and speed.

See Figure 5.2 for the Simulink implementation, and Figure 5.2 for how the bike 3D model appears. Everything shown in the plotting window was developed by Bjermeland [2006], this only shows how it responds when fed real world measurements through the Cyberbike subsystem. Note the "Substract Gravity" subsystem present in the model. This is a weak attempt at removing gravity from the acceleration data, but nowhere near an optimal solution. Section 5.10 discusses this further.

5.5 Bike Control Demo

The overall goal of this thesis, was the realization of a outdoor ride with the bike. When it became apparent that the current bike model implementation would not be able to control the bike, an alternative had to be presented. To control the bike, it has to be possible to set the steering angle, use the pendulum for balance, and apply some power to the propulsion motor. A simple approach would be to use a PID[5] controller for the steer and pendulum position and apply a constant voltage to propulsion. The control signal could be set by some external device, like a keyboard or remote controller. An attempt was made to send data from the host computer keyboard via Simulink to the bike PC.

Based on [Emanuele Ruffaldi, 2009], which is a control scheme for reading data from the keyboard into Simulink, a model which increased or decrease the reference signal based on the right and left arrow key was made. A custom tlc file had to be written for the code to compile. Unfortunately, this implementation failed miserably. The input scheme is dependent upon an open GUI window to register keystrokes, and since the code

[5] Proportional-Integral-Derivative (controller)

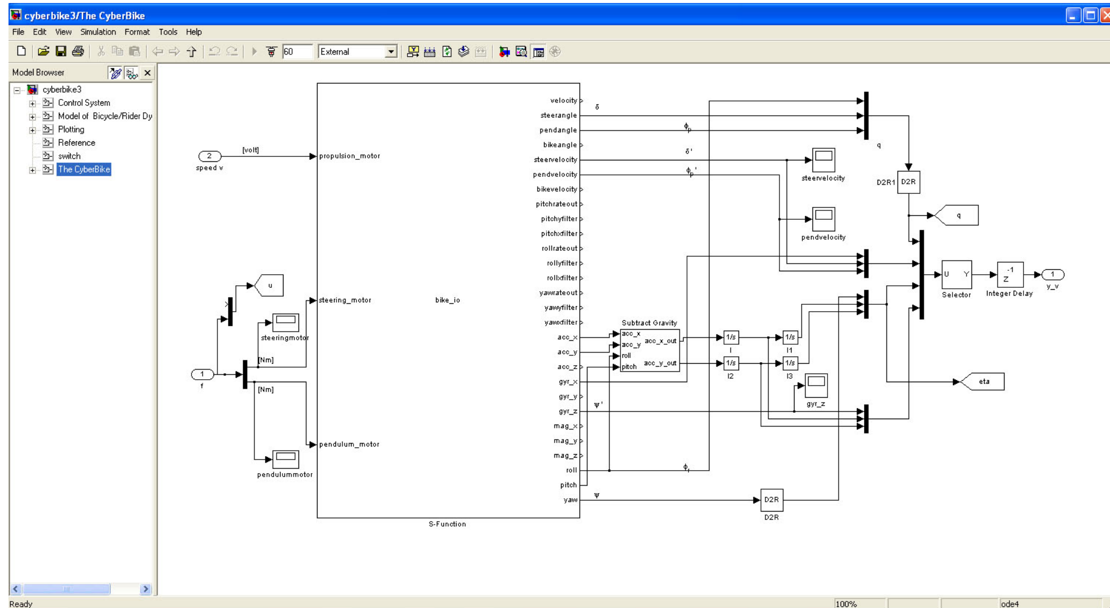

Figure 5.2: Cyberbike subsystem - the S-function.

is running on the bike PC in the terminal this does not work. Even if there was a window for accessing keystrokes, the input would have to come from the bike PC keyboard.

An alternative was attempted by connecting a constant signal to the steer and pendulum, with a tunable gain-block inserted in the signal path. The idea was to control the steer and pendulum by adjusting the gain during run-time using the mouse. This is of course a very crude and indeed inefficient way to do it, but the goal was to be able to control the bike at all. However, as it turned out, Simulink does not allow tuning of blocks during run-time when the sample time is constant, so the attempt was a failure.

Finally the idea to use the MTi as a remote controller came to mind. Using yaw, pitch and/or roll to control the steer and pendulum should be somewhat intuitive, if not directly easy. If the MTi could be connected to a laptop running QNX, and the bike accessing the MTi via the laptop using Qnet, remote control should be fairly easy to implement. First, the `devc-velo` driver was modified to make the MTi device accessible over Qnet.

What he candidate failed to remember is that the MTi device requires 5V DC, which the COM port does not provide. As a consequence, the MTi cannott be used from a remote computer, unless the power is available. Instead of pursuing electronics for something not directly applicable to the actual bike, it was decided use the MTi as a control device connected to the bike directly. The cable connecting the two is fairly long, so a test run should not require anything else for a "pipes and whistles" demonstration.

Some C code for a control system was then developed. This was two separate PID controllers for the steer and the pendulum, capable of being tuned using command line arguments. Tuning the pendulum this way was exceedingly difficult, and indeed, con-

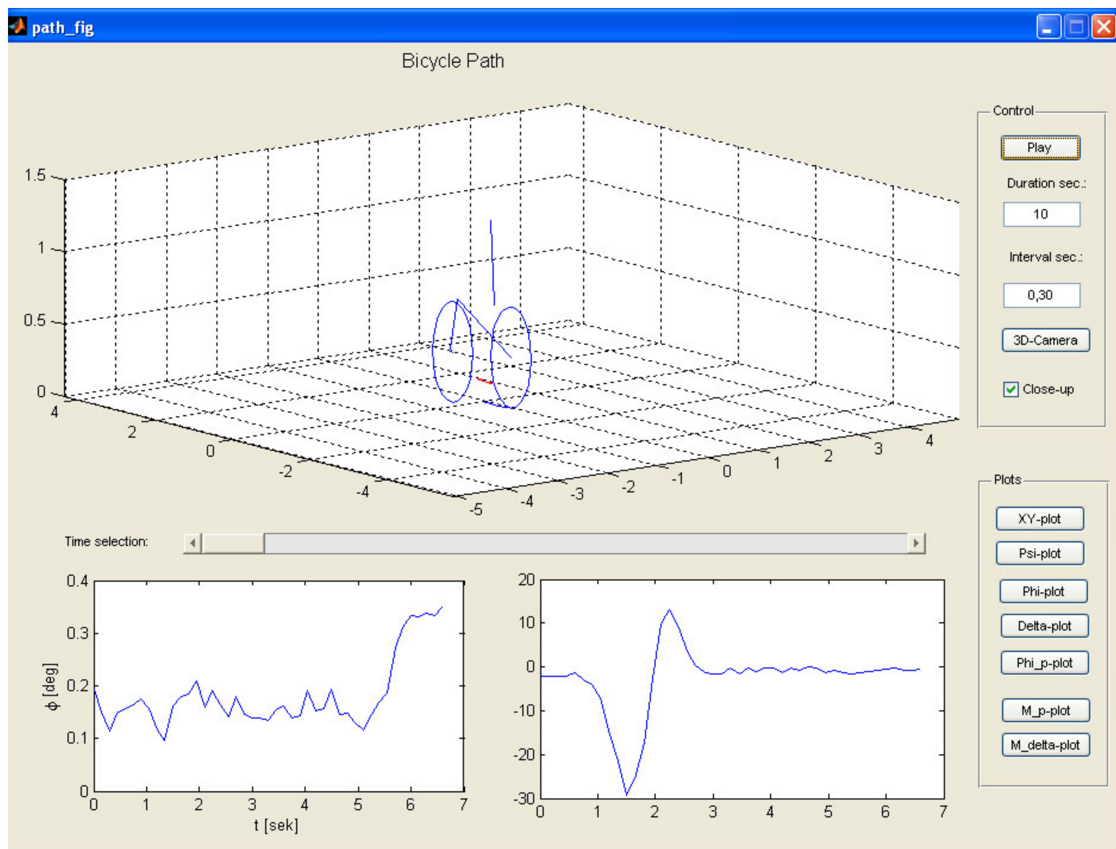

Figure 5.3: Plotting of the real world data.

trolling the pendulum based on position alone is no easy task. The code tries to scale the applied voltage based upon angle, to limit the effect of gravity. However, the code was never fully tuned; the initial values for the PID controllers are not stable. Furthermore, the control is flawed in that it does not time the control loop iteration, making the deviate and integral output incorrectly dependant upon response time of the system. Therefore, the system behaves significantly different based upon the amount of *printf()* calls used. The current implementation of the code works best if the `-v` argument is given. Printout 5.5 shows the usefile available for the executable. Using keyboard or MTi as input device, as well as disabling any of the motors is a possibility.

```
Syntax:
bike_demo [options*]
Important: devc drivers should NOT run in verbose mode.

Options:
Warning: These values may seriously affect system stability.
-q [value] Sets Kp_steer value (0.001 default)
-w [value] Sets Ki_steer value (0.00005 default)
-e [value] Sets Kd_steer value (0.005 default)
-a [value] Sets Kp_pend value (0.003 default)
-s [value] Sets Ki_pend value (0.00005 default)
-d [value] Sets Kd_pend value (0.005 default)
-g [value] Sets g_scaling value, the effect of gravity) (0.5 default)

-k Enables keyboard input, disables MTi input(reversed by default)
-m [num] Motor selection. Options:
1 - Both (Default)
2 - Steer motor only
3 - Pendelum motor only
4 - None
-v Be verbose ( -vv will give noisy mode)

Examples:
bike_demo -q 0.003 -w -0.02 -e 0.002 -vv -k
```

A Simulink "Bike Demo" was then developed when it was realized that taking the C code route is both more time consuming and harder than simply creating a Simulink model and to have RTW generate the code. Figure 5.4 shows the overall system, while Figure 5.5 shows the pendulum PID controller subsystem. Note that the reference signal is scaled and subject to saturation before reaching the controller. Here, the roll controls the the steer and pitch controls the pendulum. Even though yaw would serve as a more intuitive way to control the steer, leaving roll for the pendulum, the fact that yaw has a tendency to "wrap around", jumping from -180 degrees to 180 during operation became somewhat of a problem, and the current implementation was therefor preferred.

The "Bike Demo" model serves both as a control system for the steer and the pendulum, and as a "bare bones" implementation of the cyberbike subsystem, stripped of unnecessary files and settings. It can be used for future work if simply the interface to the bike sensors is needed, if it is shown that future developing the Cyberbike model is

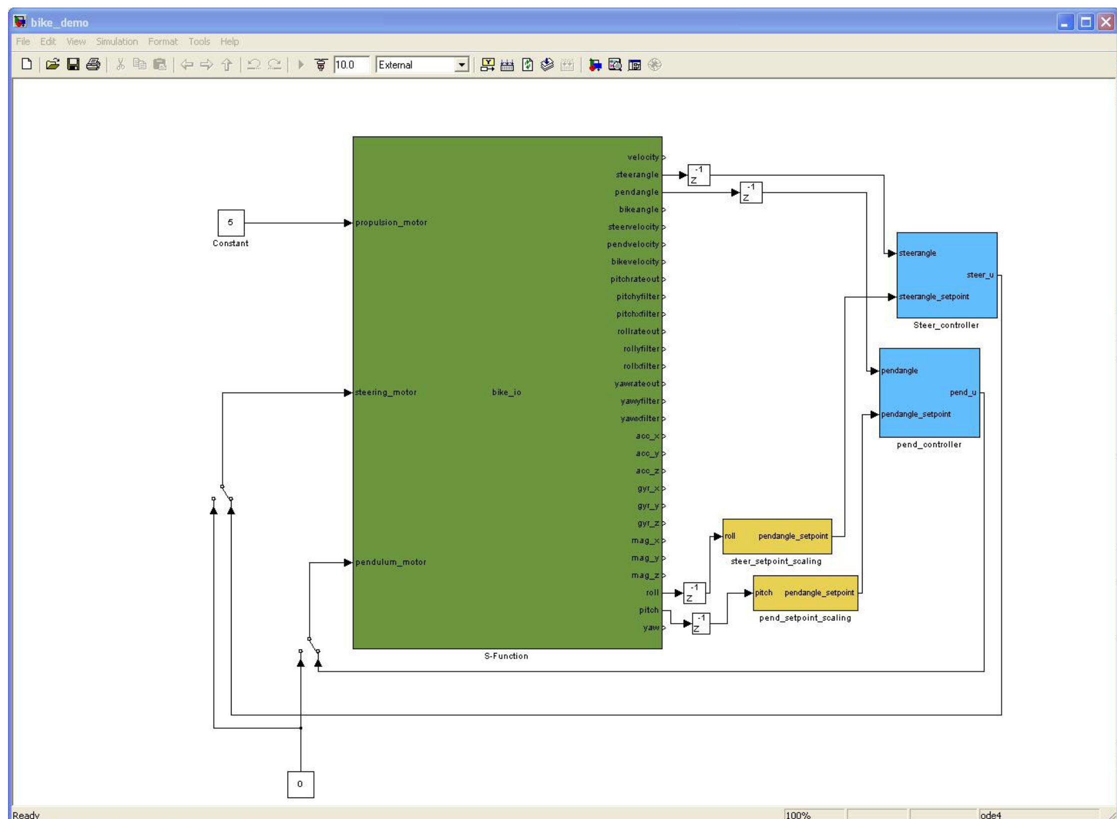

Figure 5.4: The bike demo model.

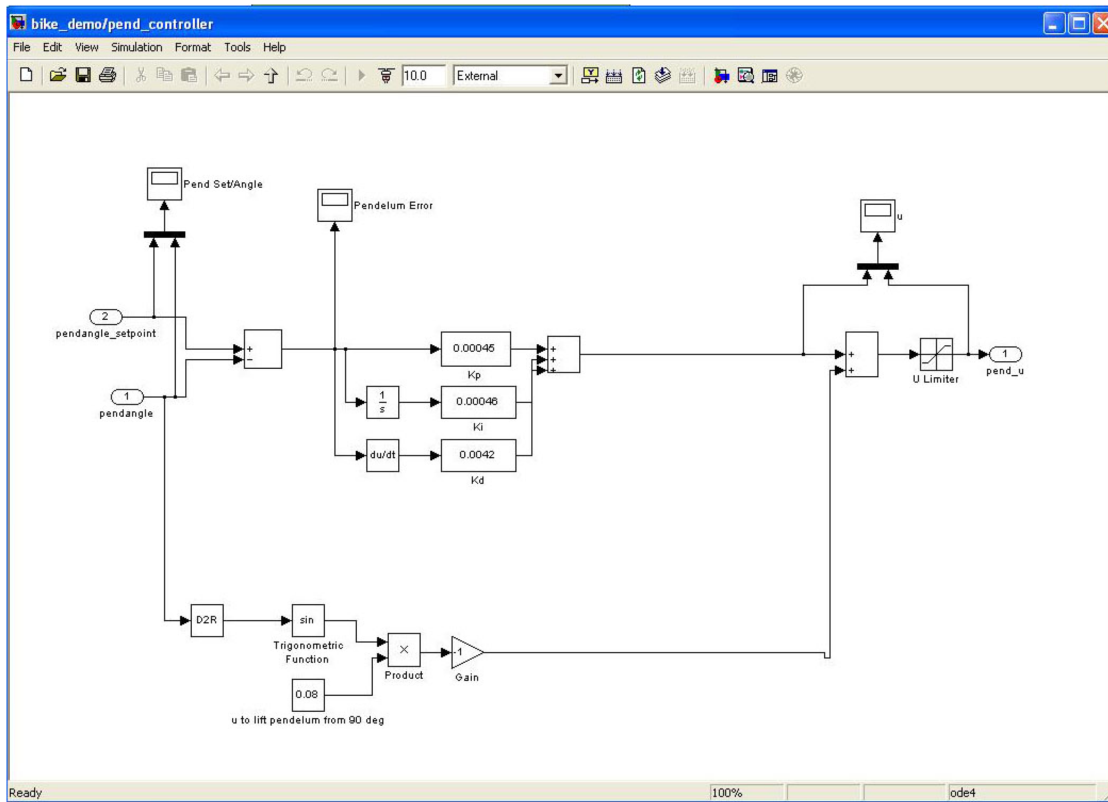

Figure 5.5: The PDI controller for the pendulum.

more hassle than it is worth. Section 6.5.2 details the behaviour of the bike demo system, and how it was tested.

5.6 Driver Enhancements

When working on a project like the autonomous bike, being able to work with and test the various drivers without the hardware available can be helpful. As such, all of the resource managers where modified to work without their hardware counterpart connected. If now hardware device is not detected, the device driver enters a "zero return mode". While in this mode, all the device files related to the driver returns 0 or the equivalent when accessed. This enables the QNX development PC to be used for testing the drivers or serve as a testing platform for the generated model code without involving the bike PC directly. Throughout the working process, this has served as an invaluable tool, and a time saver.

The various drivers has also had some of their error-return codes modified, as some of the error handling routines where returning ambiguous error messages. Overall, error handling should use more specific return codes now.

5.6.1 The Velo driver

At the start of the thesis, the `devc-velo` driver was dependant upon both `devc-mt` and `devc-imu` to run. This was determined to be redundant at this point, as the bike will most likely only need one accelerometer to operate. Therefor, `devc-velo` was modified to use the MTi as the default device, with the IMU or both available through command line options.

To accommodate the possibility of using the MTi connected to a remote computer, a command line option for setting a Qnet path for the `devc-velo` driver was implemented.

Tuning the potmeter values for the driver was originally done by adjusting the constant values defined in the driver H file. In effect, this means that whenever the potmeters become unaligned or needs readjustment, the driver has to be recompiled. A preferences file, `potmeter.cfg`, located under `/root`, was therefor created to adjust the minimum and maximum voltages for the steer and pendulum potmeters. Bias values representing the offset angle required for the angle measurement to return zero when the steer and pendulum is centered can also set through this file. This simplified the tuning process of the potmeters.

5.7 Physical Challenges and Enhancements

5.7.1 Cabinet Frame

During the course of the fall project, the autonomous bike sustained damage to the cables and cabinet, resulting in a dislodged the cabinet and unreliable connections. The bike

Figure 5.6: The bike with the new frame for the cabinet.

PC cabinet is made of a somewhat frail material, and keeping it attached to the bike with screws directly is bound to inflict future damage to the electronics. As an improvement, a frame for the cabinet to be mounted into was designed. It was made out of wood from an old pallet, painted black and attached to the bike rear frame with screws. The cabinet has a tight fit with the frame, and will remain firmly in place as long as the bicycle does not completely topple over. Figure 5.6 shows the bike with the frame attached, and the cabinet mounted.

With this construction, the cabinet can be easily removed from the bike if serious tinkering inside the cabinet is required. All the external wires where labeled with a number indicating their terminal location, making reconnecting a quicker procedure. It is no longer necessary to consult the connection tables during mounting and unmounting.

5.7.2 Potmeter strain

The potmeters on the bike led to complications. The mechanical connection between the potmeters and the drive-shaft of the steer and the pendulum motor is subject to great strain. This can be problematic when the bike becomes unstable, requiring the potmeters to be re-tuned. The screws keeping the potmeters in place where replaced to allow greater force to be applied. Unfortunately, one screw on both the steer and pendulum broke during this process. Hopefully no future adjustment has to be made to

these screws, but should the potmeters have to be removed or replaced, this could turn out to be a big problem, as special tools for screw removal is required.

5.7.3 Pendulum Limit Switches

The limit switches served more as a hindrance than anything, when trying to develop a functioning control scheme for the pendulum. The limit switches where activated when the pendulum reached +/- 17 degrees, well short of the +/- 45 range the pendulum is capable of. As the switches cut all power to the motor when activated, whenever the pendulum exceeded the limits, it would ram the pendulum frame with whatever momentum it had at the time. The switches where therefore unscrewed and left unattached to allow for greater movement.

5.8 The Propulsion motor

The autonomous bike did not have a chain connecting the rear wheel gear to the motor and the bike main gear. It turned out that the size of the gear was so big, that two chains intended for motorcycles had to be interconnected, trimmed, and used as the bike chain. However, the motor controller card is incapable of delivering sufficient current to the propulsion motor, rendering the bike unable to take an outdoor ride. Section 6.3 briefly describes how this was tested. This realization came as a hard blow during the latter parts of the thesis work, after the bike demo Simulink file had been tuned. Unable to bare the weight of the bike, the hope of first time a ride with the autonomous bike came to a full stop. A hardware solution is required to finalize this dream.

5.9 Wiring issues

Throughout the project, the motors intermittently stopped working. Probing revealed that the wires connecting the motors from the inside of the cabinet to the outside are unreliable, as poking at the wires could stop the motors from working. This problem has not been fixed. However, the problem should be viewed in connection with the , which most likely has to be replaced. In doing so, a full system check of all weiring should be preformed, and fixing the problem at this time would be work done in vain.

5.10 The MTi

The MTi had been used by another project during the spring on a remote controlled car. When it was handed over for use with autonomous bike, the serial connector at the end of the MTi cable had been replaced by a MOLEX connector. Following the Xsens [2005] guidelines, the MOLEX connector was unsoldered and removed, and replaced by the previously connected COM interface. Note that only TX, RX, VCC and GND where soldered to the MTi connector, the reminding wires where cut and left unconnected.

When the MTi was finally connected to the bike PC, the device functioned correctly, supplying the `devc-mt` driver with data. However `sloginfo -c` revealed the following continuous error: "io-char: S/W Buffer Overrun Error on /dev/ser1". This rendered the `devc-mt` unable to run in the background without being terminated by the OS, but could be circumvented by letting the driver run in the foreground in a separate terminal window. The cause of the problem is yet to be determined, but its believed to be related with the rate at which the MTi transfers data.

A separate issue, related to how an accelerometer functions, also presented itself. From Xsens [2005]:

"NOTE: The linear 3D accelerometers measure all accelerations, including the acceleration due to gravity. This is inherent to all accelerometers. Therefore, if you wish to use the 3D linear accelerations output by the MTi / MTx to estimate the "free" acceleration (i.e. 2nd derivative of position) gravity must first be subtracted."

Subtracting gravity is a non trivial problem, as the local gravity-field varies. The cyberbike subsystem in the Cyberbike Simulink model (Figure 5.2) has a block subtracting gravity based on roll and pitch. The subsystem is but a lighthearted attempt at obtaining a steady position signal after double integrating the acceleration, when the bicycle is motionless. A more throughout approach is very much required in an final control scheme.

For the MTi it is also important to be aware of how the yaw, pitch and roll values "wrap around", with the critical values being 180 and -180 degrees. Any control scheme utilizing the MTi should keep this in mind, especially considering the yaw measurement, as this is the value determine the heading of the bike. The yaw angle likely to use the whole 360 degree spectre during a ride.

5.11 The Wireless Issue

When starting the thesis, wireless connection to and from the bike was not possible. A lot of research has gone into making wireless networking work for the autonomous bike during this time. The most obvious route, and indeed the route given most attention, for wireless networking, is a USB Wireless device. One of the selling points for QNX 6.4.1 (QSSL [2009]) was the support for wireless network protocols. However, it quickly became apparent that QNX only supports a limited assortment of chip-sets, most of which are not used by USB Wifi devices available today, due to age. QSSL [2009] lists all wireless chip-sets supported by QNX 6.4.1. Ralink [2009] lists alot of wireless devices which uses the RT2500 and RT2501 chipset.

Digging around the Internet indicated that the Cisco Compact Wireless-G USB Adapter WUSB54GC-EU uses the RT2501 chipset. To make sure, Cisco tech support was contacted to verify that claim. According to Cisco, the USB device uses the Ralink RT2501USB chipset, which is listed QNX hardware database to be supported by the `devnp-rum.so` driver. In the belief that the WUSB54GC-EU would run on QNX, it was purchased. Upon delivery, the device failed to work on the QNX development PC, and

after a lengthy wait for a response on the QNX community forums, it turned out that the USB device apparently did not use the RT2501, but instead the RT3070 chip(V3). The advice given was to use the `devnp-run.so` driver. This particular driver is not, at the time of writing, available to the public. The driver was attempted to be obtain after a series of communication with the Norwegian QNX company ARX Innovation, who forwarded the driver request to the QNX headquarters in Canada. However, after some delay, they replied that the driver was "unreliable, and development had been stopped due to serious issues". In other words, the WUSB54GC could not be made to work any time soon.

In an attempt to obtain an USB dongle with a supported chip-set, all companies on the Ralink [2009] list where mailed. A simple request for them to provide the chip-set of their product, or an alternative product with one of the QNX supported chip-sets. Most companies did not reply at all, and the few that did cited "The chip-set in our products are kept secret for marketing reasons". The fact that Cisco actually gave out their information, seems, at this point, like the exception.

Following the negative response from the USB Wifi companies, the candidate was left with the unreliable Internet as the sole source of information. D-Link DWA-140 was obtained after a lengthy search seemed to indicate that it used the RT2600, available through Norwegian suppliers. This USB dongle turned out to use the unsupported RT2870 chip.

At this time, the quest for wireless networking using a USB Wifi device was abandoned. Having full network support in not necessary, as RTW support external mode using serial transfer as IPC. This means that a dedicated radio transmitter/transceiver will suffice for the control system. However, the serial transfer protocol used for external mode is only supported for Windows applications. This was realized too late in the project, so no attempt was made to port the code to the QNX platform. Wireless networking was not to be, not this time around.

Chapter 6

Tests and Experiments

This chapter describes some of the more throughout tests preformed during the work with the bike. Most of the probing with the multimeter as well as debugging software has been left out, as this is obviously integral to this type of work.

6.1 Test equipment

The multimeter used during the work on this thesis is an "FLUKE 289 True RMS Multimeter", see Figure 6.1. The multimeter has got the most usual functions expected from such a device, including voltmeter, ampere-meter, ohmmeter, DC and AC[1] setting, capacitance measurements, connectivity check ("beeping"), etc.

6.2 The Wafer-945GSE2

This section covers the test that where performed to ensure that the new motherboard was working correctly.

6.2.1 USB Ports

All four of the USB 2.0 ports where tested. This was done by connecting the mouse to each port, reboot and see if the start-up screen recognized the USB device. This was done successfully for both the on board ports, as well as the two additional ports available via the dual USB cable that was connected to the Wafer-945GSE2.

6.2.2 COM Ports

Both the COM ports where tested using the GPS device, similarly to how Sølvberg [2007, chap. 6.3.2] did it. The device was connected to a COM port, and the `cat` utility used

[1] Alternating current

Figure 6.1: FLUKE 289 True RMS Multimeter.

to investigate the `/dev/serX` file path to see if a bit stream was being received from the device. This gave positive results. The findings where future confirmed when the use of the MTi and IMU connected to both ports proved to be unproblematic.

6.2.3 Network Issues

During the course of the thesis, some trouble with the networking aspects of the bike PC appeared. During testing of the Cyberbike Simulink model, the Windows computer was unable to connect to the bike, resulting in Matlab crashing. The bike pc was at the time still able to access the Internet and the QNX development workstation. `sloginfo -c` indicated that the DHCP[2] client had been assigned a duplicate IP address. This would explain how the bike could use the net, but could not be accessed: The Windows computer was told by the DHCP sever that the IP-address belonged to a different computer. To future test the Simulink model, the problem was bypassed by setting the bike IP manually by using:

```
# ifconfig rt1 129.241.154.2/24
```

This sets the IP-address for a network interface (rt1), to the specified IP and network mask. This workaround allowed the Windows workstation and the bike PC to communicate, at the cost of Internet connectivity on the bike.

[2]Dynamic Host Configuration Protocol

This problem should be fixable by obtaining a permanent IP for the bike PC from the network administrator. However, before the candidate got around to do so, the DHCP duplicate IP issue disappeared.

6.2.4 Harddrive Issues

As described in Section 5.2, the system does not have DMA enabled. This was in spite of the fact that every possible hard drive option in the BIOS related to DMA was tested by selecting an option, rebooting and loading the `.boot` file with DMA enabled. Every single alternative resulted in a frozen login screen/system crash. Unable to determine the cause of the problem, the bike system now has DMA disabled by default.

6.3 Motor tests

After the chain had been mounted on the bike, the propulsion motor was tested to see if it could handle the strain of pulling the wheel load. Applying 5V by the use of the `echo` tool, the motor managed to make the rear wheel spin. The current indicator on the power source peaked at about 14 A for a short while, before stabilizing around 8 A. If the wheel was subject to hardly any resistance at all (like the tip of a shoe), the motor would stop. The propulsion motor is rated at 18A, but since the Baldor TFM 060-06-01-3 is only capable of delivering 6 A continuously, the motor controller card was identified as the culprit. Unless this problem is fixed, the bike will not be able power itself forward.

Both the steer and propulsion motor functioned perfectly during tests with the `echo` tool.

6.4 Batteries stress test

When starting the thesis, the batteries where completely depleted. The workshop had to charge the batteries for several weeks, before they regained their former 12V potential. To test if the batteries would still be able to deliver sufficient current to the system, the batteries where connected and the system booted. The motors where then started one by one, finally engaging everything at once. No problem seemed to arise; the system behaved identically to using a power generator. At this point, the batteries are given a clean bill of health.

6.5 Simulink tests

This section covers the testing that was performed on the existing Simulink model when the thesis started, and the Bike Demo model that was developed during the thesis work.

6.5.1 Cyberbike model tests

The Cyberbike Simulink model was extensively tested to determine why it would terminate after only a few time-steps. The model did originally use a fixed time-step of 0.02 seconds. [Sølvberg, 2007, chap.6.6] suggested that the cause of the problem could be related the old motherboard being too slow, so this was the first part to be examined. By running the bike drivers on the QNX workstation in "zero-return mode", 2.0 GHz of processing power was available; a significant increase over the measly 400 MHz of the Wafer 971A. It was quickly determined however, that model communicating with the Windows computer running Simulink still terminated only a few time-steps after being started. The error produced was "Memory fault (core dumped)". This gave reason to believe that the problem was not, at least primarily, related to processing speed.

To determine the root of the problem, a lot of debugging messages where added to the core C files generated by RTW, trying to track down exactly where the memory fault appeared. Unfortunately, it became apparent that the exact point of failure was hard to track down, as the RTW framework is dependant upon a vast amount of files. Keeping it simple (stupid), quickly presented itself as a better alternative: Disable the S-function code, and see if the problem disappeared. This was done by simply stopping the device files from being sampled. And indeed, now the code initiated, executed and terminated properly. At this time it was attempted to sample just one device, but this resulted in an error message. To see if increasing the sample time could rectify the problem, the fixed time-step was at first set to 0.25 seconds. Now the model ran smoothly, successfully terminating when stopped through Simulink. At this point, all device files where enabled, and, by trying to bound the sample time from below, 0.15 seconds was determined as a stable value, constantly running without failure.

Throughout the testing process, several bugs with the `bike_io_wrapper.c` code where fixed, some described in Section 5.4.

6.5.2 Bike demo tests

The bike demo Simulink model was tuned and tested for stability. This was done separately for the steer and the pendulum by starting in centered position, with the angle close to zero, and set the reference to the face left after five seconds. Figure 6.2 shows how the steer position follows the reference for the final tuning, while Figure 6.3 shows the same for the pendulum.

The stability of the control scheme was determined to be satisfactory as a "proof of concept" demonstration. The steer and pendulum should follow the reference to such a degree that a operating the two is possible within reasonable limits.

6.6 Device Peripherals and associated drivers

All the drivers where, at the minimum, modified to work even with no hardware connected, so everything had to be properly tested for any bugs or errors with and without a

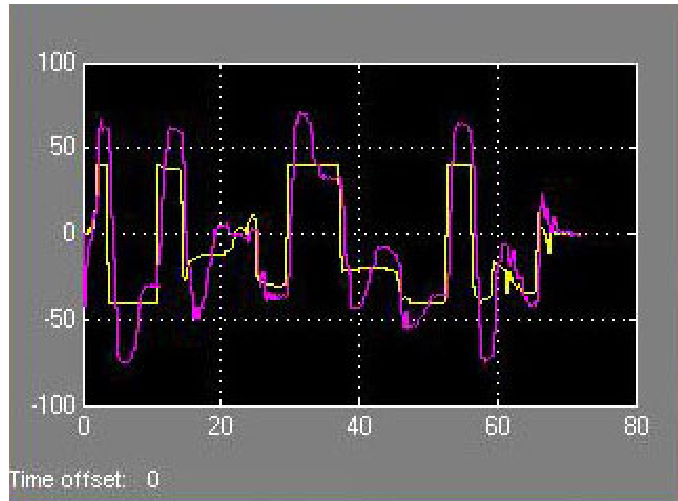

Figure 6.2: Steer referance vs. steer position.

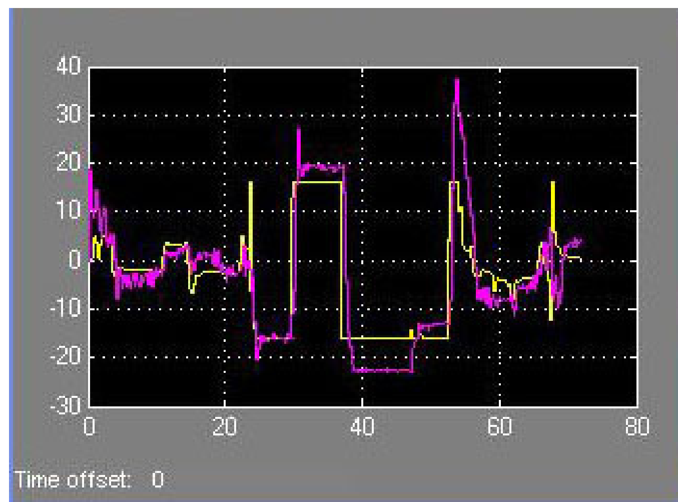

Figure 6.3: Pendulum referance vs. pendulum position.

device connected. The `cat` utility was used to access every device file for a given driver, and see if the output was as expected. This way the *read()* and *open()* functions where accounted for. For all but the the `devc-velo` driver, the `devc-velo` device files where used to test the various *devctl()* for the drivers, as `devc-velo` uses *devctl()* to access the underlying resource managers. The `echo` utility was used to test the *write()* functions. All tests where preformed with the driver set to noisy mode, to provide as much debug information as possible. The `devc-velo` was tested with custom code that performed *open()*, *read()*, *write()* and *devctl()* calls on the velo device files, and printed the results. All of these tests where completed successfully.

6.7 GPS

The GPS driver, with the GPS connected, gave numerous errors and warning during operation. Upon inspection, it was determined that this was not a physical issue, as the data packets received over the serial port adhered to the GPS documentation. The GSV messages revealed that the GPS had 12 satellites in view. However, the GSA messages indicated that no fix was possible. This was compliant with the status of the LED on the GPS which had a solid red light; indicating that no fix has been made (as opposed to a flashing light, indicating a fix).

It is believed that the electromagnetic noise and concrete walls in the development area prevented the GPS from working properly. A possible solution could have been to put the GPS device outside the window, but the way the device is installed onto the bike cabinet, makes removing it hard. As the tests implies that the GPS device is still functioning correctly, and a fix is possible environment permitting, further work interested in utilizing the the device should be possible.

Chapter 7

Further Work

Typically, the chapter describing alternatives for future work is reserved for later. However, as the autonomous bike project has come to be a large and complex system, there are so many minor and major tasks to be accomplished that a simple afterthought about the work ahead would not serve the project well. This chapter seeks to recap the problems remanding to be tackled and to highlight critical issues that should be fixed before moving ahead. Additionally, this chapter will point out larger tasks waiting to be undertaken, ideas for expanding the bike horizons, task perhaps serving as master theses in their own right.

7.1 Recap of Existing Problems

This section repeats the problems that are unfixed, which where described Chapter 5, and possible work that can be done to rectify them.

7.1.1 The first journey

The first outdoor ride with the bike was stopped short by the limits of the Baldor TFM 060-06-01-3, described in Section 5.8. In that regard the most pressing issue would be to replace the motor controller card, or amplify the power output to allow the propulsion motor enough power to pull the bike. It would be natural to test and fix the motor wires covered in Section 5.9 at the same time. If these two issues are corrected, there should be nothing preventing the bike from using the Bike Demo Simulink model to control the bike using the MTi, and taking the first ride outdoors.

7.1.2 Minor Issues

Below is a listing of minor issues that should be dealt with as time permits:

- The screws fixing the potmeters in place are damage. The university workshop can probably be very helpful in removing the broken screws. However, this will not change the fact that the mechanical strain will eventually lead to the potmeters having to be re-tuned. Fortunately, tuning the potmeters is now relatively easy, as it can be done as described in Section 4.3.5.
- If the .tmf file is used in future work, would be advisable to investigate why the Makefile is compiling using -ansi and not -std=c99 as an option. The file works completely as is, but coding using -std=c99 might be preferable.
- The sample time used in the Simulink models is now relatively high at 0.15 second. Sampling fewer device files in the S-function might allow this interval to be shortened.
- The `devc-velo` driver is currently assuming that torque produced by the motors is the voltage applied multiplied by ten. Improving this relationship would be a good idea, moving forward with a more complex control scheme.
- Implementing a velocity control scheme for the bike would be highly beneficial and would probably serve stability, especially for the pendulum, well. This is not necessarily a trivial task, but should be investigated.
- The inside of the bike cabinet is somewhat of a mess. Lose wires and cables are a hazard to the system, and the metal floor inside could accidentally connect or short wires touching it. A major "spring cleaning" could serve the cabinet well.
- The dual USB cable connected to the Wafer-945GSE2 is lose, as the mount point for the previous USB cable had different dimensions. Accommodating a new mounting point for the cable could be beneficial.
- The Ethernet port available on the outside of the cabinet is believed to be damaged. Plugging directly into the Wafer-945GSE2 works perfectly, but using the cabinet port indirectly sometimes disrupts the network. Either the wires should be fixed, or the port replaced.
- The GPS device is believed to be fully functional, but unable to fix its location, due to the development environment. An outdoor test of the GPS device should be preformed, to determine if it is indeed functional.
- The MTi driver is plagued by an stack overflow issue. As a result it cannot be run in the background using the modifier. It is believed to be caused by high transfer rate. Lowering the rate of transmission requires a write function to be made for the MTi driver, for device to recieve commands externally.
- The pendulum limit switches has been unscrewed and remind loose. An alternative mounting point should be considered, or even a different approach altogether. Having some sort of hardware limiter for the pendulum however, is important, both for

the preservation of the motor and for human safety. Implementing some additional limit into the software drivers could also be advisable.

- A new release version of QNX, version 6.5.0, is available. Upgrading the bike OS should be done to benefit from whatever upgrades the new version provides.

7.1.3 Wireless networking

Section 5.11 describes how a lengthy attempt at making Wifi available for the bike was failed to be realized by the use of a USB dongle. Should the `devnp-run.so` driver ever be released, than the WUSB54GC-EU USB Adapter should be capable of providing Wifi. No research has to been done regarding QNX 6.5.0 and how it relates to wireless USB devices. The new version can possibly provide a solution.

Alternatively, the RTW code for serial IPC could be ported to QNX, and the bike controlled by the use of a radio transmitter/transceiver. There are PC/104 cards that available that can sever this function, so some research should be put into exploring this alternative.

The final, and perhaps easies solution would be the acquisition of a WiFi bridge. A WiFi bridge can be connected directly to the ethernet port of the Wafer-945GSE2, and requires no drivers to work. The challenge of using a bridge is finding a device small enough that it can be fitted on the bike, and that is not too expensive. A way to power the device should also be explored. The candidate recommends checking out WiFi bridges made for gaming devices like the Xbox or Wii for alternatives fitting these criteria.

7.2 Moving Forward

This section covers some of the broader tasks that can be undertaken to further the bike project. [Sølvberg, 2007, Chap. 8] discusses interesting alternatives, some of which will be summed up here.

7.2.1 Simulink model

At this time, the Simulink model developed by [Bjermeland, 2006] does not integrate well with the cyberbike subsystem. Assumptions and simplifications done for the model, renders the bike unstable with the current control scheme. A deeper diagnostic of the existing model could be performed, but without taking a better look at the underlying theory, the time might be better spent elsewhere. [Lebedonko, 2009] takes a more comprehensive look at bike modelling, and one would be well advised to take his master thesis into consideration if a new model is to be constructed. If a model could be created, based upon the measurements available through cyberbike subsystem, a truly autonomous is a possibility. A challenge in this regard would be to implement a intuitive way for a bike operator to set the reference signal for the bike, through Simulink. Handling the MTi

acceleration data should also put up for consideration, and how best to subtract gravity from the data, as well as a method for handling angles "wrapping around" from -180 to 180 degrees. A way for the model to use the GPS would be optimal. An extract from the MTi documentation ([Xsens, 2006]) is presented here for inspiration:

"It's possible to double integrate accelerometer data, after proper co-ordinate transformations and subtraction of the acceleration due to gravity, to obtain 3D position data. To implement this in a practical implementation some issues will be encountered: 1) You will need a 'starting point', a reference 3D position, from which you can start to integrate the 3D acceleration data. 2) Noise on the acceleration data (about 1 mg RMS) and small offset errors and/or incorrectly subtracted acceleration due to gravity, will be integrated and over time will cause huge (drift) errors in the position and velocity estimate. The conclusion is that it depends very much on the (type) of motion you want to register if this approach is feasible. Typically, short duration movements, preferably cyclical, with frequent known reference positions will work well."

Line of Sight

[Sølvberg, 2007, Chap. 8.3] mentiones how a Line of Sight algorithm could be used with the bike, as suggested by [Bjermeland, 2006]. This involves defining waypoints for the bike, while using a a control algorithm for making the bike take the shortest path to the destination. For this purpose, the GPS device should come in handy.

7.2.2 Optimizing `devc-dmm32at`

[Sølvberg, 2007, Chap. 8.4] explains how the DMM-32-AT driver can be utilized to save on processing power. With the introduction of the Wafer-945GSE2, processing power is abundantly less scarce, so the priority of this task should be considered low.

7.2.3 Videocamera

[Sølvberg, 2007, Chap. 8.5] mentions that the installation of a camera for recording the bike's point of view would be interesting. As the development of the autonomous bike comes along, this also becomes more relevant. The system has access to four times the processing power it once did, making this option very plausible to implement. However, focusing on implementing a wireless connection should be prioritized, as a camera mounted on a bike requiring an Ethernet cable seems redundant at best.

7.2.4 Mechanical Brakes and Overall Safety

Today, the bike is not safe for public demonstration. This is not only caused by an unstable controllers or unreliable wires; the autonomous bike cannot break. One alternative could be to mount regular bike breaks, but this might turn out to be difficult, as the bike-frame dates back to the eighties, with none of the parts adhering to today's bike

standards. An alternative could be to install a motor for controlling something imitating human legs; a way for the bike to force a dead stop through fiction with the ground.

Additionally, the gears of the bike are fully exposed, and is a safety hazard in their own right. Some protective housing could be beneficial, to prevent fingers from snapping.

7.2.5 Overall system

Moving forward, some way to power up the system and automatically have all drivers and applications necessary to control the bike start should be implemented. Currently, a lot of tinkering is needed to prepare the bike for external communication. As most of the underlying drivers should work properly at this point, focusing on making the autonomous bike a polished and solid system is possible.

Chapter 8

Discussion

Working on the autonomous has presented a varied assortment of challenges, some of which where dealt with satisfactory, and some of which arguable could have been handled better. This chapter discusses how the work could have been done differently, and reflects upon choices made throughout.

8.1 Choice of new motherboard

The Wafer-945GSE2 was chosen, as it represented a significant upgrade over the old hardware, as described in Section 5.1. The cost, about 3000 NOK, was comparable to similar SBC on the marked. The board has preformed well throughout the work, the only persisting problem relates to DMA having to be disabled. This might just as well be caused by the new hard drive. The board was bought outside the normal avenues used by the University, as it was easier to have the costs reimbursed than to wait for purchase approval. This means that the MVA had to be paid in full, which could have been avoided. However, as the candidate already was late starting the project, it was decided to acquire a motherboard as quickly as possible.

8.2 Choice of storage medium

The reason for choosing a hard drive instead of a flash card was twofold. Firstly, when the old Eurobot HD died, a replacement had to be found quickly. Previous theses gave the impression that working with flash cards was riddled with problems, and to avoid unnecessary complications these where avoided. Secondly SSD drives, containg no moving parts was also an alternative, but these are significantly more expensive than a regular HDD. [Sølvberg, 2007, Chap. 7.1] discusses the problems related to hard drives of the type chosen: the mechanical parts of the drive are subject to external disturbances, very much prevalent in the bike system. Should the bike topple over, the contents of the hard drive could be permanently damage. However, so far the hard drive has functioned

satisfactory, and unless experience indicates otherwise, it might serve its purpose for the duration of the first outdoor ride.

8.3 Unresolved problems

Chapter 7 covers a lot of the issues that remain in the system to this day. One could argue that some of these problems are so minute that they should have already been handled. However, when working on a complex system, such as the autonomous bike, alone, issues has to be prioritized: there are only so many work hours to go around. As the main focus of the work was to get the bike out the doors and sent on a first time ride, some corners where cut to move the project along. When the ride never materialized, one could argue that all the issues should have been fully resolved before moving on to the next task. However, it was not always possible to fix a problem immediately, an therefor other tasks where undertaken in the mean time, sometimes uncovering yet different issues to be resolved. Simply working through the issues is not always straightforward or even possible.

The candidate very much shares the sentiment expressed by [Sølvberg, 2007, Chap. 7.4]: *"A human weakness is that they make mistakes"*. The act of probing a physical system of this nature can in an of itself lead to problems; writing software code can result in minor bugs or even hardware damage. However, a more methodical approach to the system, especially during the earlier work with the bike should have been contemplated. Time and time again, it became apparent that breaking problems down to linearly dependant tasks, cut down on the time needed for development and testing. Early on, when getting familiarized with the system, aimless actions with no real rhythm or reason where preformed, sometimes causing more harm than good. More thought should have been put into the *way* the work was being preformed. Being methodical is key.

8.3.1 Lack of Wireless Networking

The failure to introduce wireless networking to the bike project was a hard burden to bare. The problem alone consumed far to much time considering no actual solutions where presented. Scourging the Internet for tidbits of information about various chipsets and USB devices is time that could have been well spent elsewhere. The QNX community forums have very little traffic, and posts made there where very slowly responded to, if at all. Combined with mailing uncooperative corporations in vain, these actions took a heavy toll on morale. If nothing else, the challenge has thought the candidate how fickle consumer electronics can be, if one wishes to detail the gritty bitty insides. Hopefully options presented in Section 7.1.3 can make further attempt easier to accomplish.

8.4 Reflection

Very much like [Sølvberg, 2007], the errors inherent in the bicycle system where larger than expected at the starting point of the thesis work. Overall, the limiting factor throughout was time, as problems starting piling up and system components, previously tested and given a clean bill of health, started to fail. At times it feels like the same tasks has to be redone time and time again, for no apparent reason. With the benefit of hindsight, where the correct decisions made?

As mentioned above, the work put into USB Wifi devices, seemed like a waste of time. Had the realization that Wifi bridges where an option come earlier, perhaps the time spent on the dongles could have used to modify a bridge to comply with the physical system. As for the failure to conduct a outdoor ride with the bike, the candidate is not as disappointed. Even if it had been know that the motor controller card was insufficient to power the propulsion motor, it is not a given that simply replacing it would not have cut into some of the other tasks that actually where finalized during the work. The fact that a functional bike demo model has been implemented, and a final Simulink interface to the hardware has been completed, is just as important for the completion of the bike system as a functional propulsion motor.

Things take time. Every little problem that arises can turn out to require hours of work, and working completely isolated on a complex system can sometimes be a daunting task. Optimally, students continuing the work on the bike should work in pairs, as running ideas and experiments past a fellow student before jumping to action be quite valuable.Hopefully, the issues still persisting in the bike system has been fully documented throughout this thesis, serving as both a sign of warning and a source for inspiration for kindred spirits embarking on the journey that is the autonomous bike.

Chapter 9

Conclusion

The idea of an autonomous bicycle, dating back to the early eighties, is starting to gain momentum. The goal is to have a bike take a ride outdoors, without the assistance of a rider. To achieve this, balance is provided by an inverted pendulum, controlled by a motor. Yet other motors control the steer and provide propulsion, everything controlled by a onboard computer interfacing with various measurement devices. The dream is to have a complex computer model control these devices, resulting in a stable, riderless bicycle.

The main part of the thesis consisted of finalizing the hardware and software parts of the system, all of which where believed to be near completion at the onset of the project. Part of the assignment was to determine and solve the problems preventing the system from conducting a first journey outside.

This first ride was not realized, but came immensely close. A "proof of concept" control system has been developed, to allow the steer and pendulum to be controlled by the MTi accelerometer device. The overall instrumentation system has been improved, with a new motherboard and hard drive added to the system. The physical frame supporting the bike cabinet has been constructed and mounted. It should make further work easier, as the cabinet can easily be dismounted. All the drivers have been modified to function without their respective hardware device connected, and overall the software system has been made more stable and reliable. The cyberbike subsystem available through Simulink has been fully implemented and tested, and should serve as a easy way for more complex models to interface with the bike hardware in the future.

Arguably, too much time was spent trying to introduce wireless networking to the system, by the use of USB Wifi devices. QNX turned out to provide little in the way of support of new USB Wifi dongles, and tracking down supported chip-sets was determined to be nigh impossible. Hopefully, the fruitless work done to make the USB devices work, has revealed compelling alternatives for further work on the project.

The thesis itself can be viewed as reference documentation for the hardware and especially, software, as a go-to point for new students starting work on the autonomous bicycle. Including further work as a part of the main text was intentionally done to

allow a quick-start, point-by-point guide to the work required to finalize the project. The appendix provides a step-by-step guide for making the bike demo work, making it possible to see the bike in action early, to get a sense of what the system can do.

Bibliography

ACT Batteries. [online], 2007. URL http://www.actbatteries.co.uk/. [Accessed 30 Dec 2009.

Baldor ASR. *Pulse Width Modulated Transistor Servodriver TFM Instruction Manual.* Baldor ASR GmbH, Dieselstraße 22, D-8011 Kirchheim-M¬Π$nchen, WestGermany, 1988. Version1 : 11/07/88$.

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

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.

DtC-Lenze as. [online], May 2007. URL http://www.dtc.no. [Accessed 25 Dec 2009].

Emanuele Ruffaldi. Simulink keyboard input v2. [online], 2009. URL http://www.mathworks.com/matlabcentral/fileexchange/24216. [Accessed 1 June 2010].

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

IEI Technology Corp. *WAFER-945GSE2 3.5" SBC, User Manual, Rev 1.0*, March 2009.

IEI Technology Corp. WAFER-945GSE2 Image. [online], 2010. URL http://t0.gstatic.com/images?q=tbn:l1XoU1_8q9GKKM:http://www.nmr-corp.com/product/cat09/images/entry001.jpg&t=1. [Accessed 20 August 2010].

Vegard Larsen Lasse Bjermeland and Pål Jacob Nessjøen. Using simulink real time workshop™ to create programs for qnx™platform. Technical report, ITK, NTNU, Trondheim, Norway, Feb 2007.

Igor Olegovych Lebedonko. Autonomous bicycle. Master's thesis, Telemark University College (NTNU), Telemark, June 2009.

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.

QNX Software Systems. *Programmer,Äôs Guide*. QNX Software Systems GmbH Co, 6.3 edition, 2009a. URL `http://www.qnx.com/`. [online].

QNX Software Systems. *System Architecture*. QNX Software Systems GmbH Co, 6.4.1 edition, 2009b. URL `http://www.qnx.com/`. [online].

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. Qnx product documentation. [online], 2009. URL `http://www.qnx.com/developers/docs/index.html`. [Accessed 20 June 2010].

Ralink. Ralink rt2500 chipsets based wireless 802.11g devices. [online], Feb 2009. URL `http://ralink.rapla.net/`. [Accessed 20 May 2010].

Samsung. HD322HJ 320 GB/7200RPM/16M Image. [online], 2010. URL `http://forum.ge/uploads/post-54-1272375242.jpg`. [Accessed 20 August 2010].

Audun Sølvberg. Cyberbike. Master's thesis, Norges teknisk-naturvitenskapelige universitet (NTNU), Trondheim, June 2007.

The MathWorks, Inc. Real-time workshop user's guide. [online], 2009. URL `http://www.mathworks.com/access/helpdesk/help/toolbox/rtw/ug/ug_intropage.html`. [Accessed 15 Dec 2009].

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`.

Appendix A

Contents on DVD

- code: Contains all the code for the bike drivers, as well as some support programs.
- matlab: Files needed for RTW.
- Simulink: Contains the bike demo, and cyberbike Simulink model, along with some older versions.
- Report: The latex files used for this thesis.
- Documents: Contains datasheets, manuals and documentation.
- Velo CD: Backup of previous master theses work. Files of interest might be the test programs available for some of the drivers.

Appendix B

How to start the bike 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.
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-945GSE2 (SBC).
6. Connect a keyboard into the K connector on the mouse and keyboad cable connected to the Wafer-945GSE2.
7. Plug an ethernet cable into one of the RJ-45 connecters on the Wafer-945GSE2. The RJ-45 on the cabinet is damaged, and should not be used.
8. Connect a mouse (if wanted) into one of the USB connector of the Wafer-945GSE2. Alternatively, the dual USB cable connected to the Wafer-945GSE2can be used.
9. Make also sure that the hard drive are connected to one of the SATA connectors 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. The device cables should be labled. Only the MTi is required for the control system to work, as the IMU is somewhat redundant. The IMU is also hardcoded to use COM2. The GPS device has been shown to not work indoors. 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. However, the motor controller card cannot continuously provide more than 6A to any one motor. 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[1] 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). If the development PC used during this thesis is available, the login is user: "root" with no password, similarly to the bike PC.

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
    ```

    each in a seperate terminal window (only devc-mt *requires* a seperate terminal, as it cannot be run in the background). For a typical test run, only `devc-dmm32at`, `devc-mt` and `devc-velo` has to be started. The `devc-velo` has to be started last. Device files should be accessible from the `/dev/` directory when the drivers are successfully started.

17. The easiest way to share files between computers is to work out of your sambaad domain. The domain can be mounted in QNX by typing:

    ```
    # fs-cifs -l //sambaad.stud.ntnu.no:/username /home
    ```

[1] Integrated Development Environment

in a terminal window. This will prompt for username and password and then mount your sambaad files in the `/home` directory. If the computers used for this thesis are used, it is sufficent to mount the disk on the QNX development workstation. The bike PC can access the dev PC files through the `/net/dev` directory.

18. Assuming the Windows workstation used for this thesis is available, login using your own user name and password on the win-ntnu-no domain. On the C: drive there should be a folder labeled `CYBERBIKE`. Inside, a `cyberbike` folder and `bike_demo` folder can be found.

19. Mount your sambaad domain, and copy the `bike_demo` folder over to your domain.

20. Open the `bike_demo` folder copied (now working in the mounted folder), and load the `bike_demo` .mdl found inside the `bike_demo` folder.

21. Hit ctrl+B when the model has loaded. This will initiate the Real-Time Workshop build process.

22. On the QNX workstation, locate the `bike_demo` folder you copied to your domain. All your files should be accessible from the `/home` folder.

23. In the terminal, positioned inside the `bike_demo` folder, type:

    ```
    # cd bike_demo_qnx_rtw
    # make -f bike_demo.mk
    ```

    This will build an executable, which will be located in the `bike_demo` folder.

24. On the bike PC, locate the `bike_demo` folder using the terminal, by accessing the development PC through:

    ```
    # cd /net/dev
    ```

25. Take note of the bike PC IP address using:

    ```
    # ifconfig
    ```

26. Start the executable with:

    ```
    # ./bike_demo -tf inf -w
    ```

    The bike PC will now await a start signal from Simulink from the Windows workstation.

27. On the Windows computer, in the Simulink model locate Real Time workshop settings from pull down menu. Click "Interface". In the MEX-file arguments field, type in 'BIKE_IP', in the same way the IP appearing in the field already has been. Most likely only the last part of the IP will differ (or it might even be the same). Click OK.

28. Hit the connect button in the Simulink window. This will initiate external communication with the bike. When the Play button becomes clickable, start the model. It is now possible to control the steer and pendulum using the MTi through roll and pitch.

Appendix C

Connection tables

This appendix provides tables covering most of the connections in the bike. The tables where in full created by [Sølvberg, 2007] and are identical to the ones in his master thesis, but has been included here for consistency and ease of use.

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

C.2 Baldor

			Pin row - c			Pin row - a		
	Axis	Motor	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 C.3: Connection table for Baldor TFM 060-06-01-3

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

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