NTNU
Norwegian University of
Science and Technology

GPS Guided R/C Car
The Local Bug Test Platform

Peder Wenstad

Master of Science in Engineering Cybernetics
Submission date: July 2010
Supervisor: Bjørnar Vik, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem Description

The Local Hawk project is continuous student project with the goal to develop an autonomous unmanned aerial vehicle. Student from several educational institutions collaborate and develops parts of the total system trough student projects, summer projects and master theses. Before implementing guidance systems on an actual Unmanned Aerial Vehicles it is desirable to test the algorithms on a safer platform.

The overall goal of this thesis is to develop a test platform called Local Bug. It is supposed to be based on a Radio Controlled (R/C) car and use electronics developed in earlier Local Hawk projects. The platform needs to be capable to log data during tests, making it possible to evaluate the controller and guidance system performance.

When designing control systems it is important to be able to test the performance in a simulator before applying it on the real system. Such a simulator is to be developed by implementing a vehicle model.

It is in the interest of the Local Hawk project to be able to generate C-code from control systems designed in SIMULINK. Real-Time Workshop is able to generate generic C-code from SIMULINK models. The student is to consider the possibility of using this approach and evaluate the result if it is used.

The Local Bug platform is to be used to test controllers and guidance systems.

As this assignment is a part of a larger project, other students should be able to continue this work by reading the report.

Assignment given: 15. February 2010
Supervisor: Bjørnar Vik, ITK

Abstract

This thesis is a part of the Local Hawk student project where the overall goal is to develop a Autonomous Unmanned Aerial Vehicle (AUAV). The project was initiated by Kongsberg Defence Systems (KDS) and is developed in collaboration with the Norwegian University of Science and Technology (NTNU). In an AUAV it is necessary to have a guidance system in order to be autonomous. To be able to test guidance principles in practice without risking the Local Hawk airframe the need of a ground based test platform became apparent. This thesis is the development of the Local Bug Test Platform.

The Local Bug is based on a R/C car and utilizes electronics used and developed in the Local Hawk project. In order to understand the system behaviour, vehicle modelling is discussed and two models are presented. One of them is used in a SIMULINK simulator design for the Local Bug. The simulator is used to ensure the correct controller behaviour before it is tested on the real system.

Phoenix II is the backbone of the Local Bug electronics and is used to gather measurement data, log data to memory and execute controller algorithms. Custom made C-code is used as a framework providing all the functionality needed. A introduction to Real-Time Workshop is given, and a step-by-step guide on how to use RTW generated C-code on the Phoenix II for controller purposes.

A heading controller is designed and used in combination to two different guidance algorithms. The Line of Sight algorithm aims for the next waypoint regardless where the vehicle is positioned relative other waypoints. A Cross-Track Error algorithm with look ahead distance functionality focuses on minimizing the vehicles distance from a desired path. Both algorithms successfully guides the vehicle trough the test route visiting all the waypoints.

The Local Bug is now usable as a test platform for controller algorithms. Measurements are logged to memory making it possible to analyse the performance after a test has been conducted. The overall functionality is found to be satisfactory and the Local Hawk project now has its desired controller/guidance test platform.

Preface

The Local Hawk is a ongoing student project, where students work independently on different topics. The overall goal is to develop an Autonomous Unmanned Areal Vehicle (AUAV) by unifying the student's contributions. New tasks and areas of interest are found during the process and new tasks are given in order to consequently move towards the main goal.

Students working on the Local Hawk should be able to use earlier reports as a basis for new development. A great deal of effort should be put into documenting the tasks performed in such a way that new students can utilize the results without having to conduct extensive research on topics already covered by the overall project.

This thesis is supposed to be read from beginning to end, and later chapters are based on theory presented earlier. The mathematical preliminaries section is added so inexperienced readers will not have to read mathematical theory elsewhere. It is more of a listing of theorems and rules than a theory presentation, for more information see the citations given. The Real-Time Workshop section in Chapter 4 can be read stand alone and be used as a step by step guide to get controller functionality from SIMULINK and onto a real-time application developed in C-code.

In addition to this thesis, the reader should have a look at the CD attached. Testing the simulator and viewing the videos can help the reader to better understand the concepts presented. Both guidance blocks presented is included and the reader can switch between them and simulate different test routes by changing the waypoints. MATLAB/SIMULINK needs to be installed on the computer used to be able to use the simulator.

I would like to thank: Bjørnar Vik for being my supervisor and for providing good advice when needed. KDS and Jon Bernhard Høstmark for their dedication to the Local Hawk project and providing the possibility of taking part in the development of something as cool as autonomous vehicles. Per Magnus Veierland for providing the *beta* drivers for Phoenix II and helping me understand how they where supposed to be used. Last but not least my son and my loving fiancée for providing motivation and support through the process of this thesis writing.

Contents

Mathematical Preliminaries

The Sine Rule

Angles in a triangle are related to the length of their opposite side as follows (Rottmann, 2003):

$$\frac{\sin\alpha}{a} = \frac{\sin\beta}{b} = \frac{\sin\gamma}{c} \tag{1}$$

where a, b, c, α, β and γ are defined in Figure 1.

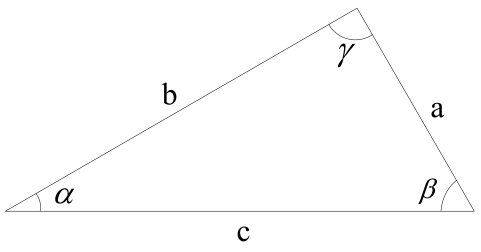


Figure 1: Triangle with sides a, b and c; and angles α, β and γ.

Rotation Matrices

> *A rotation matrix* $\boldsymbol{R}$ *satisfies:*
>
> $$\boldsymbol{R}\boldsymbol{R}^\top = \boldsymbol{R}^\top\boldsymbol{R} = \boldsymbol{I}$$
>
> $$det\boldsymbol{R} = 1$$
>
> *which implies that* $\boldsymbol{R}$ *is orthogonal. Consequently, the inverse rotation matrix is given by:*
>
> $$\boldsymbol{R}^{-1} = \boldsymbol{R}^\top$$
>
> Fossen (2002)

Throughout this thesis the following notation is used for rotation matrices:

- $\boldsymbol{R}^a_b$ is the rotation matrix from coordinate frame b to a.
- $\boldsymbol{R}_{k,\alpha}$ is a rotation matrix representing a rotation α degrees about the k-axis.

Several rotations can be combined in one rotation matrix by multiplying the rotations in the right order. Rotation matrix c to a is the product of the rotation matrices from c to b and b to a.

$$\boldsymbol{R}^a_b \boldsymbol{R}^b_c = \boldsymbol{R}^a_c \tag{2}$$

The following notation is adopted when transforming a vector from one coordinate frame to another

$$\boldsymbol{v}^a = \boldsymbol{R}^a_b \boldsymbol{v}^b \tag{3}$$

Since the inverse of a rotation matrix is equal to its transposed the inverse transformation is

$$\begin{aligned} \boldsymbol{v}^b &= \boldsymbol{R}^{a\top}_b \boldsymbol{v}^a && (4) \\ &= \boldsymbol{R}^b_a \boldsymbol{v}^a && (5) \end{aligned}$$

Arc length

The arc length l of a part of a circle is

$$l = \alpha r \tag{6}$$

where α is the angle(in radians) of the circle arc to be found and r is the radius of the circle.

The nth-Degree Taylor Polynomial

Suppose that the first n derivatives of the function $f(x)$ exists at $x = a$. Let $P_n(x)$ be the nth-degree polynomial

$$\begin{aligned} P_n(x) &= \sum_{k=0}^{n} \frac{f^{(k)}(a)}{k!}(x-a)^k \\ &= f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(n)}(a)}{n!}(x-a)^n. \end{aligned} \tag{7}$$

Then the values of $P_n(x)$ and its first n derivatives agree, at $x = a$, with the values of f and its first n derivatives there. This theorem and more information on Taylor series can be found in Edwards and Penny (2002).

Abbreviations

2D	Two Dimentional
3D	Three Dimentional
AFCS	Automatic Flight Control System
AUAV	Autonomous Unmanned Aerial Vehicle
CG	Centre of Gravity
CO	Centre of Origin
DOF	Degrees of Freedom
ECEF	Earth-Centered-Earth-Fixed
GPS	Global Positioning System
HiBu	Buskerud University Collage
KDS	Kongsberg Defence Systems
LQR	Linear Quadratic Regulation
NED	North-East-Down
NTNU	Norwegian University of Science and Technology
P	Proportional
PI	Proportional-Integral
R/C	Radio Controlled
ISA	Inertial Sensor Assembly
SMCU	Servo Control and Measurement Unit
SPI	Serial Peripheral Interface
TWI	Two Wire Interface
UAV	Unmanned Aerial Vehicle
UiA	University of Agder

1 Introduction

Local Hawk is a continuous student project involving the making and development of an Autonomous Unmanned Aerial Vehicle (AUAV). Focusing on developing solutions other students can build upon and use for bachelor/master theses. The project was initiated by Kongsberg Defence Systems (KDS) in 2008 in collaboration with the Norwegian University of Science and Technology (NTNU) and is based on earlier work on AUAVs by several students from NTNU. The initiative was taken by KDS to boot the interest for flying objects in the academic circle. In addition, demonstrate that solid theoretical work in synergy with a practical task increases the motivation and learning outcome for the student.

During the summer 2008, five students built a basic platform for an AUAV by modifying a model aircraft and applying basic functions to enable automatic flight control(Miljeteig et al., 2008). They succeeded in mounting an Inertia Measurement Unit (IMU), a servo controller and a radio link communicating through a common Two Wire Interface bus (TWI). The project was a success, but due to time limitations several tasks and improvements remained undone. Further developments have been conducted through the NTNU course *Interdisciplinary Teamwork* and some master and bachelor theses.

The Kongsberg summer project 2009(Hagen et al., 2009) initiated the development of new and more compact electronics. Veierland (2010) completed this work trough a bachelor thesis. This resulted in Phoenix II, an electronic platform designed to implement all functionality needed by autopilot software to control the AUAV. Sensor modules, processing nodes, high-capacity storage, and radio communication capabilities are included on this platform.

Different guidance principles were evaluated and tested in a SIMULINK simulator by Vold (2009). A safe platform to test different controllers and guidance algorithms in practice without putting the Local Hawk airframe at risk, was now needed.

The main goal of this thesis is to develop such a platform based on a R/C car and hardware used in earlier Local Hawk projects. Guidance principles evaluated by Vold (2009) are to be implemented and tested. This thesis is the first practical application using the Phoenix II, since it has been partially developed simultaneous to this work. A secondary goal of this thesis is to be able to use Real-Time Workshop to generate generic C-code from SIMULINK models and use this code with the Phoenix II.

This thesis takes the reader from mathematical vehicle modelling, via controller design, to practical application on a physical system. Chapter 2 presents different vehicle models and discusses the pros and cons of using them to implement a simulator. One of the models are implemented in SIMULINK and simulator for the Local Bug is introduced. A more practical approach is taken in Chapter 3 as the different hardware components used on the Local Bug and their corresponding drivers/software are presented. Chapter 4 presents the controllers and guidance schemes to be used on the Local Bug platform. This chapter also provides instructions on how to generate C-code from SIMULINK models with the help of Real-Time Workshop and use this code in addition to custom made C-code. The guidance algorithms and controller are tested on the platform and the results are presented in Chapter 5. In Chapter 6 a discussion is made, while the conclusion and reflections on possibilities of further development of the platform can be found in Chapter 7.

A CD is attached to this thesis in Appendix C. It contains the Local Bug simulator, a video presenting some of the results of this thesis, test logs and videos corresponding to the tests presented in Chapter 5 and the source code developed and used on the Phoenix II.

2 Vehicle Dynamics

A vehicle model is used to simulate behaviour of a vehicle and has a tool to design controllers. There are several ways to model an automobile, were different models have different complexity and accuracy. The complexity ranges from 14 Degrees of Freedom (DOF) 3D models to pure kinematic bicycle models in 2D. This thesis scope is set on the motion in the xy-plane, and hence there is no need for 3D models.

This chapter presents a rigid body model and a kinematic model and discusses which model to use as a base for the further development of the Local Bug. In addition a vehicle model is implemented in SIMULINK to be used to simulate the Local Bug behaviour. For information on car modelling in general and alternative vehicle models to the ones presented in this thesis, see Tjønnås and Johansen (2010), Ackermann (1993), Setiawan et al. (2009) and Petersen (2003).

2.1 Axis systems

The Earth Centred Earth Fixed (ECEF) frame can be used to describe positions and motions relative to the surface of the Earth. This reference frame has its origin in the centre of the Earth. The z-axis coincides with the Earth's rotation axis while the x-axis points towards the intersection of $0°$ longitude and $0°$ latitude. In order to make the reference frame Earth Fixed it rotates about the z-axis with the angular velocity to make the x and y axis stationary with respect to the surface of the Earth. GPS provides position described in Cartesian coordinates in the ECEF frame. However, since this is a rather non-intuitive interpretation of the ECEF coordinates they are usually transformed into ellipsoidal coordinates (longitude, latitude and height).

The *North-East-Down*(NED) frame is defined relative to the Earth's reference ellipsoid. It neglects the curvature of the earth and is therefore only valid in for a limited area close to the centre of origin. This coordinate system is the tangent plane to the reference ellipsoid. The x-axis points towards true north, the y-axis towards east and the z-axis points downwards perpendicular to the Earth's surface (the reference ellipsoid).

The *body*-axis system is defined as shown in Figure 2.2. This frame has origin in the vehicle's centre of mass. The x-axis points forward, the y-axis points to the left side of the vehicle and the z-axis points upwards. For more information on these axis systems, see Vik (2009).

The wheel frame is fixed, with the origin at the centre of the wheel and the x-axis pointing forward. Positive wheel spin, that is wheel spin causing the vehicle to move forward, is defined as a positive rotation about the y-axis pointing to the left. As a result of the right hand rule the z-axis points upwards, giving us the positive steering rotation when turning left.

2.2 Planar Rigid Body Model

The idea behind the planar model is to model the vehicle as a box moving on a horizontal surface. This approach simplifies the model considerably, however when considering the heading dynamics this model can be found satisfactory. A rigid body can move in six degrees of freedom; however planar models only include three of them. The translation in the x and y direction, and a rotation about the z-axis. Since the roll and pitch are neglected there is no need to include the shock absorber dynamics.

Figure 2.1 illustrates the vehicle and its body frame b relative to an inertial frame i. $\vec{r}_o$ is the position of the Centre of Origin (CO) in the body frame. $\vec{r}_g$ is the Centre of Gravity (CG) and is given by

$$\vec{r}_g = \vec{r}_o + \vec{r} \tag{2.1}$$

where $\vec{r}$ is the vector from CO to CG. $\boldsymbol{r}^b$ is the body frame coordinate vector for $\vec{r}$ and can be rotated to the inertial frame by

$$\boldsymbol{r}^i = \boldsymbol{R}_b^i \boldsymbol{r}^b \tag{2.2}$$

The velocities of o and g are given by

$$\vec{v}_o = \frac{{}^i d}{dt}\vec{r}_o, \quad \vec{v}_g = \frac{{}^i d}{dt}\vec{r}_g \tag{2.3}$$

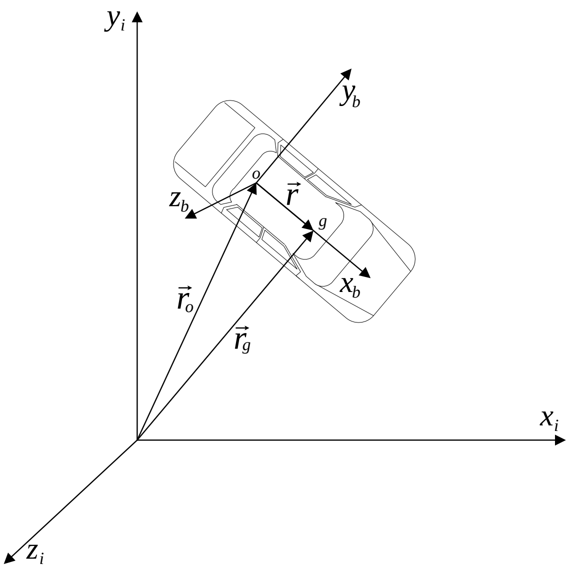


Figure 2.1: The chassis with its body frame and centre of gravity. The subscript i denotes the inertial frame while b denotes the body frame.

According to Egeland and Gravdahl (2002) time differentiation of equation (2.1) in a moving reference frame gives

$$\vec{v}_g = \vec{v}_o + \frac{{}^b d}{dt}\vec{r} + \vec{\omega}_{ib} \times \vec{r} \tag{2.4}$$

For rigid bodies the CG is fixed to the body, that is

$$\frac{{}^b d}{dt}\vec{r} = \vec{0} \tag{2.5}$$

which gives

$$\vec{v}_g = \vec{v}_o + \vec{\omega}_{ib} \times \vec{r} \tag{2.6}$$

Newton's second law states(Tipler and Mosca, 2004):

> *The direction of the acceleration of an object is in the direction of the net external force acting on it. The acceleration of an object is proportional to the net external force $\vec{f}_{net}$, in accordance with $\vec{f}_{net} = m\vec{a}$. The net force acting on an object is the vector sum of all forces acting on it: $\vec{f}_{net} = \sum \vec{f}$. Thus,*
>
> $$\sum \vec{f} = m\vec{a}$$
>
> Newton's second law

For rotational motion the same principals apply, but with inertia and moments instead of mass and force. Newton's second law for translational motion and rotational motion can be presented as

$$\sum \vec{f} = \frac{\mathrm{d}}{\mathrm{d}t} m\vec{v} \tag{2.7}$$

$$\sum \vec{m} = \frac{\mathrm{d}}{\mathrm{d}t} I_g\vec{\omega} \tag{2.8}$$

where $\vec{f}$ is the coordinate free force vector acting on the centre of gravity and $\vec{v}$ is the velocity of the CG with respect to the inertial frame. Equation (2.7) and (2.8) can be rewritten and expressed in the body frame as

$$\begin{aligned}
\vec{f}_{net} &= \frac{{}^{i}d}{dt}(m\vec{v}) \\
&= \frac{{}^{b}d}{dt}(m\vec{v}) + \vec{\omega}_{ib} \times (m\vec{v}) \\
&= m(\dot{\vec{v}} + \vec{\omega}_{ib} \times \vec{v}) \\
\boldsymbol{f}^{b}_{net} &= m(\dot{\boldsymbol{v}}^{b} + \boldsymbol{\omega}^{b}_{ib} \times \boldsymbol{v}^{b}) \qquad (2.9)
\end{aligned}$$

$$\begin{aligned}
\vec{m}_{net} &= \frac{{}^{i}d}{dt}(\boldsymbol{I}\vec{\omega}_{ib}) \\
&= \frac{{}^{b}d}{dt}(\boldsymbol{I}\vec{\omega}_{ib}) + \vec{\omega}_{ib} \times (\boldsymbol{I}\vec{\omega}_{ib}) \\
&= \dot{\vec{\omega}}_{ib} + \vec{\omega}_{ib} \times (\boldsymbol{I}\vec{\omega}_{ib}) \\
\boldsymbol{m}^{b}_{net} &= \boldsymbol{I}\dot{\boldsymbol{\omega}}^{b}_{ib} + \boldsymbol{\omega}^{b}_{ib} \times \boldsymbol{I}\boldsymbol{\omega}^{b}_{ib} \qquad (2.10)
\end{aligned}$$

Figure 2.2 illustrates the force, moment and kinematic vectors for the rigid vehicle in the xy-plane. These vectors given in the body frame are:

$$\boldsymbol{f}^{b} = \begin{bmatrix} f_x \\ f_y \\ 0 \end{bmatrix}, \qquad \boldsymbol{v}^{b} = \begin{bmatrix} \dot{v}_x \\ \dot{v}_y \\ 0 \end{bmatrix}, \qquad \dot{\boldsymbol{v}}^{b} = \begin{bmatrix} \dot{v}_x \\ \dot{v}_y \\ 0 \end{bmatrix} \tag{2.11}$$

$$\boldsymbol{m}^{b} = \begin{bmatrix} 0 \\ 0 \\ m_z \end{bmatrix}, \qquad \boldsymbol{\omega}^{b}_{ib} = \begin{bmatrix} 0 \\ 0 \\ \omega_z \end{bmatrix}, \qquad \dot{\boldsymbol{\omega}}^{b} = \begin{bmatrix} 0 \\ 0 \\ \dot{\omega}_z \end{bmatrix} \tag{2.12}$$

Since the body axes coincide with the axes of the principal coordinate frame it is reasonable to assume that the vehicle has a diagonal moment of inertia matrix.

$$\boldsymbol{I}^{b} = \begin{bmatrix} I_x & 0 & 0 \\ 0 & I_y & 0 \\ 0 & 0 & I_z \end{bmatrix} \tag{2.13}$$

Substituting (2.11)-(2.13) into the equations of motion (2.9)-(2.10) provides the

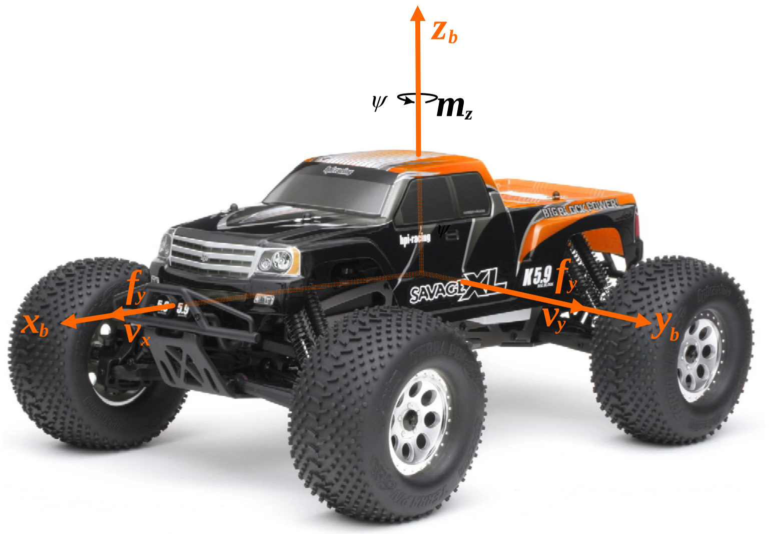


Figure 2.2: The force, moment and kinematic vectors in the xy-plane presented in the body frame.

following equations:

$$\begin{aligned}\boldsymbol{f}^b_{net} &= m(\dot{\boldsymbol{v}}^b + \boldsymbol{\omega}^b_{ib} \times \boldsymbol{v}^b) \\ &= m \begin{bmatrix} \dot{v}_x \\ \dot{v}_y \\ 0 \end{bmatrix} + m \begin{bmatrix} 0 \\ 0 \\ \omega_z \end{bmatrix} \times \begin{bmatrix} v_x \\ v_y \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} m\dot{v}_x - m\omega_z v_y \\ m\dot{v}_y + m\omega_z v_x \\ 0 \end{bmatrix} \end{aligned} \tag{2.14}$$

$$\begin{aligned}\boldsymbol{m}^b_{net} &= \boldsymbol{I}\dot{\boldsymbol{\omega}}^b_{ib} + \boldsymbol{\omega}^b_{ib} \times \boldsymbol{I}\boldsymbol{\omega}^b_{ib} \\ &= \begin{bmatrix} I_x & 0 & 0 \\ 0 & I_y & 0 \\ 0 & 0 & I_z \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ \dot{\omega}_z \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \omega_z \end{bmatrix} \times \left(\begin{bmatrix} I_x & 0 & 0 \\ 0 & I_y & 0 \\ 0 & 0 & I_z \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ 0 \end{bmatrix} \right) \\ &= \begin{bmatrix} 0 \\ 0 \\ I_z\dot{\omega}_z \end{bmatrix} \end{aligned} \tag{2.15}$$

The only non-zero equations are the two first Newton equations (2.14) and the third Euler equation (2.15). They affect the vehicle in the x and y direction and about the z-axis, that is in the 3DOF included in the planar model. Hence the planar rigid body equations of motions are(Reza, 2009):

$$f_x = m\dot{v}_x - m\omega_z v_y \tag{2.16}$$

$$f_y = m\dot{v}_y + m\omega_z v_x \tag{2.17}$$

$$m_z = \dot{\omega}_z I_z \tag{2.18}$$

2.2.1 Planar wheel dynamics

The wheels are the system input, affecting the planar vehicle motion in all degrees of freedom. For a planar model the suspension is neglected since it only affects the vehicle in roll and pitch. Figure 2.3 illustrates the left front wheel and its wheel frame. As described in Section 2.1 the x_w and y_w axis are fixed to the wheel and rotated about the z axis with a steering angle δ_{fl} compared to the body frame. f_{xfl} and f_{yfl} are the forces caused by the wheel decomposed into vectors parallel to the body axis.

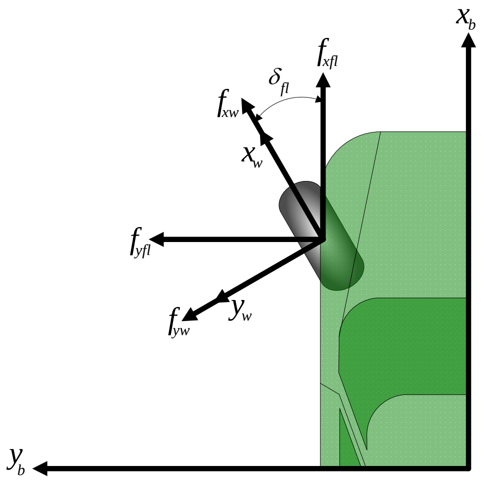


Figure 2.3: The front left wheel frame and forces caused by the wheel

The turning of a car is caused by the wheels, when the steering wheel turns the wheels get misaligned with the direction of travel. This gives rise to a friction force in the lateral direction of the wheel. This force is illustrated in Figure 2.3 as f_{yw}. Decomposed into body coordinates f_{yw} gives a negative contribution to the forward force f_{xfl}.

Consider a rear wheel driven vehicle, the rear wheels will create a force causing the vehicle to accelerate forwards. The negative forward force from the front wheel with a non-zero steering angle δ will decrease the acceleration in the x_b direction and redirect the force form the rear wheels to the x_w direction causing the vehicle to alter course. A four wheel driven vehicle will basically have the same behaviour.

However the front wheels will provide thrust in the x_w direction given, by the steering angle, affecting the vehicle in both x and y direction accordingly.

For a planar model the suspension is neglected since it only affects the vehicle in roll and pitch. Figure 2.3 illustrates the left front wheel its wheel frame. As described in Section 2.1 the x_w and y_w axis are fixed to the wheel and rotated about the z-axis with a steering angle δ_{fl} compared to the body frame. f_{xfl} and f_{yfl} are the forces caused by the wheel decomposed into vectors parallel to the body axis. The translational forces caused by wheel i:

$$\begin{aligned}\boldsymbol{f}_i^b &= \boldsymbol{R}_w^b \boldsymbol{f}_i^w \\ &= \begin{bmatrix}\cos\delta_i & -\sin\delta_i \\ \sin\delta_i & \cos\delta_i\end{bmatrix}\begin{bmatrix}f_{x_i} \\ f_{y_i}\end{bmatrix} \\ &= \begin{bmatrix}f_{x_i}\cos\delta_i - f_{y_i}\sin\delta_i \\ f_{x_i}\sin\delta_i + f_{y_i}\cos\delta_i\end{bmatrix}\end{aligned} \tag{2.19}$$

These forces will also affect the rotational motion because they affect the body in CG(origin of the body frame). Hence, there exists a non-zero vector $\boldsymbol{r}_i$ from the CG to the centre of each wheel frame. The moment contribution generated by wheel i is:

$$\begin{aligned}\boldsymbol{m}^b &= \boldsymbol{R}_w^b \boldsymbol{m}^w + \boldsymbol{r}_i^b \times \boldsymbol{f}_i^b \\ &= \boldsymbol{R}_w^b \boldsymbol{m}^w + \boldsymbol{S}(\boldsymbol{r}_i^b)\boldsymbol{f}_i^b \\ &= \begin{bmatrix}0 \\ 0 \\ m_{z_i}\end{bmatrix} + \begin{bmatrix}0 & -z_i & y_i \\ z_i & 0 & -x_i \\ -y_i & x_i & 0\end{bmatrix}\begin{bmatrix}f_{bx} \\ f_{by} \\ 0\end{bmatrix} \\ &= \begin{bmatrix}0 \\ 0 \\ m_{z_i}\end{bmatrix} + \begin{bmatrix}0 \\ 0 \\ -y_i f_{bx} + x_i f_{by}\end{bmatrix}\end{aligned} \tag{2.20}$$

Hence, the total force and moment contribution from the wheels are:

$$\sum_i f_x = \sum_i f_{x_i}\cos\delta_i - \sum_i f_{y_i}\sin\delta_i \tag{2.21}$$

$$\sum_i f_y = \sum_i f_{x_i}\sin\delta_i + \sum_i f_{y_i}\cos\delta_i \tag{2.22}$$

$$\sum_i m_z = \sum_i m_{z_i} + \sum_i (x_i f_{by} - y_i f_{bx}) \tag{2.23}$$

2.2.2 Tyre dynamics and friction model

Pacejka (2006) divides the tyre models into four categories. The first category is models obtained from experimental data only. Measured tyre characteristics are used to construct tables and interpolation schemes or mathematical formulas are used to form the models. The form of the formulas is usually assessed by the regression methods used to find the parameters giving the best fit to the measured data.

The second category is called the similarity approach. Basic characteristics typically obtained from measurements are connected and modified to include off-nominal relations. Models within this category are useful in applications were real-time computations are required.

Simple formulation of physical models may provide sufficient accuracy for limited fields of application (the third category). Simplifications are made to keep the formulas manageable; however they still need to include the significant matters to be tenable. These models can provide great understanding of the tyre dynamics.

The models in the final category are found from complex physical modelling. These models are primarily used for more detailed analysis of the tyre.

Aligning Moment Tyre Model

Reza (2009) proposes to use a simplified tyre model where the lateral force caused by the tyre friction only depends on the wheel side slip angle α. Figure 2.4 illustrates a wheel with the corresponding reference frames and angles. β is the vehicle side slip angle, that is the angle defined as the rotation from the body frame to the vehicle velocity vector $\boldsymbol{v}$. The steering angle is the angle between the body frame and the wheel frame and is denoted by δ. Defining the wheel side slip as the rotation from the wheel frame to the vehicle velocity vector yields the relation between α, β and δ:

$$\alpha = \beta - \delta \tag{2.24}$$

This model is very simple and is based on defining the lateral forces to give rise to a *aligning moment* trying to reduce α. The aligning moment affects the vehicle about the z-axis and tends to align the x_w-axis with the vehicle velocity vector for small α. Reza (2009) defines the lateral forces as:

$$f_y = -C_\alpha \alpha \tag{2.25}$$

where C_α is the *cornering stiffness* defined as:

$$C_\alpha = \lim_{\alpha \to 0} \frac{\partial(-f_y)}{\partial \alpha}$$

The *aligning moment* about the z-axis is caused by the lateral force at the centre of the wheel frame.

2.3 Kinematic Single Track Model

Rajamani (2006) describes a Single Track Model, also called bicycle model. This model merge the effect of both front and both rear wheels into one front and

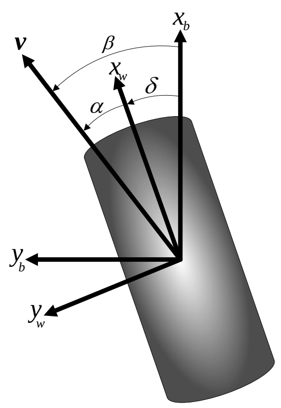


Figure 2.4: The slip angle denoted by α. w and b denotes the wheel frame and the body frame respectively.

one rear wheel placed on the body x-axis with equal distance from the y-axis as the original wheels. By applying this simplification this model resembles a bicycle model, hence the second name. A Kinematic Single Track Model which describes the vehicles motion without considering the forces affecting the vehicle. The kinematic equations of motion are based purely on the system's geometric relationships.

Figure 2.5 illustrates a Single Track Model with the wheels at point A and B. l_f and l_r are the distance from the centre of gravity to the front and rear wheel respectively. The vehicle's heading is denoted as the angle ψ from the body x-axis to the inertial x-axis. The point O is origin of the circular motion of the vehicle. Assuming that the velocity vectors at each wheel points in the direction of the corresponding wheel, O is defined as the intersection between the lines drawn perpendicular to the two wheels. This assumption is equivalent to assuming the slip angles at both wheels to be zero and is a reasonable assumption for wheel-based vehicles at low speed. R is the radius of the centre of gravity's circular motion about O, hence it is the length of the line OC. The velocity vector at the centre of gravity is perpendicular to OC. Its direction with respect to the body x-axis is called the side slip angle β.

The following derivation follows that of Rajamani (2006). Application of the sine rule, defined in Equation (1), to triangle OCA yields:

$$\begin{aligned}\frac{\sin(\delta_f-\beta)}{l_f} &= \frac{\sin(\frac{\pi}{2}-\delta_f)}{R}\\ \frac{\sin\delta_f\cos\beta-\sin\beta\cos\delta_f}{l_f} &= \frac{\cos\delta_f}{R}\end{aligned} \tag{2.26}$$

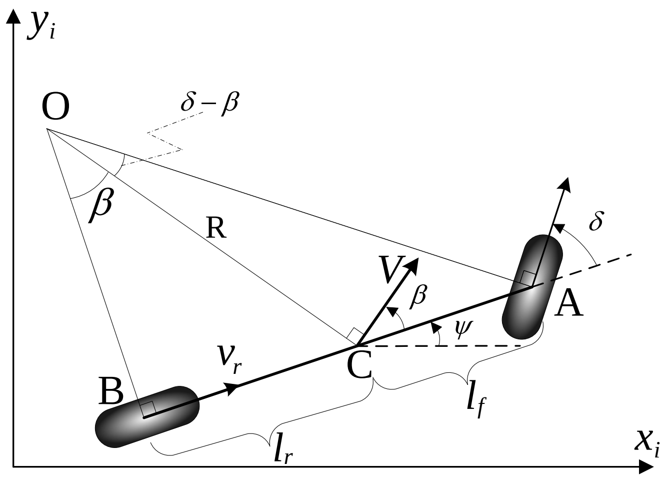


Figure 2.5: Kinematic Single Track model

The lack of rear wheel steering simplifies the equation for OCB:

$$\begin{aligned}\frac{\sin\beta}{l_r} &= \frac{\sin\frac{\pi}{2}}{R}\\ \frac{\sin\beta}{l_r} &= \frac{1}{R}\end{aligned} \tag{2.27}$$

Multiplying both sides of Equation (2.26) by $\frac{l_f}{\cos(\delta_f)}$ and (2.27) by l_r yields Equation (2.28) and (2.29) respectively.

$$\tan\delta_f\cos\beta - \sin\beta = \frac{l_f}{R} \tag{2.28}$$

$$\sin\beta = \frac{l_r}{R} \tag{2.29}$$

Due to low vehicle speed it is reasonable to assume the radius of the vehicle path R to be slowly varying. The change in orientation and the angular velocity about O is therefore approximately equal. Hence,

$$\dot{\psi} = \frac{V}{R} \tag{2.30}$$

Combining Equation (2.28)-(2.29) and substituting Equation (2.30) for R yields:

$$\dot{\psi} = \frac{V\cos\beta}{l_f + l_r}\tan\delta_f$$

The vehicle velocity in the inertial frame can be found by projecting the velocity vector onto the inertial x and y-axis respectively. The angle between the inertial frame and the vector is defined as $\gamma = \psi + \beta$. Hence, the equations of motion are given by:

$$\dot{x}_i = v \cos \gamma \tag{2.31}$$

$$\dot{y}_i = v \sin \gamma \tag{2.32}$$

$$\dot{\psi} = \frac{V \cos \beta}{l_f + l_r} \tan \delta_f \tag{2.33}$$

The side slip β can be obtained by multiplying Equation (2.28) by l_r and substitute it into Equation (2.29) multiplied by l_f

$$\beta = \arctan \left(\frac{l_r \tan \delta_f}{l_f + l_r} \right) \tag{2.34}$$

2.4 Choice of Model

The planar rigid body model is based on the forces and moments affecting a mass. This model is quite intuitive and it is easy to understand how different forces will affect the vehicle. Disturbances can be implemented and since the model design is modular it is possible to switch part of the model with more accurate models. The complexity of the model is mostly dependent on the wheel and friction model. These models also have a great influence on the accuracy of the total model. With the right wheel/friction model the planar rigid body model includes effects caused by wheel spin and road conditions.

The Kinematic Single Track Model is far more limited. It is based on the system's geometric relationships and is derived with the assumption that a wheel only moves in the direction it is pointing. Wheel spin and slippery road conditions are therefore not possible to include unless the model is totally altered. The model equations are not intuitive without knowing the geometric aspect they are derived from. As long as the assumptions hold it is undoubtedly easier to get accurate simulations than if the rigid body model is used.

When choosing which model to implement in SIMULINK and use to test and simulate different controllers it is important to consider under what circumstances the vehicle is supposed to operate. In this case the goal is to test guidance algorithms which are not affected by the surrounding environment. The obvious choice of test area would be a relatively flat area with limited amount of obstacles and unforeseen disturbances. There is no need to provoke wheel spin when testing a guidance algorithm since the sliding motion should be controlled by a heading controller. Under these circumstances the obvious choice is the Kinematic Single Track Model since none of its disadvantages takes affect in the test environment. It is therefore no reason to implement a more complex model to obtain a result almost equal to the one provided by the single track model.

2.5 The Local Bug Simulator

Simulation is the imitation of something real, like a process state or the motion of a vehicle. The act of simulating something generally entails representing certain key characteristics or behaviours of a selected physical or abstract system.

The simulator presented in this section uses the Kinematic Single Track Model presented earlier. It is supposed to represent the behaviour of the Local Bug vehicle and can be used to test different controllers without having to put the actual system at risk.

2.5.1 SIMULINK Implementation of the Vechile Model

Figure 2.6 shows the implementation of the Kinematic Single Track Model(Eq. 2.31-2.34). Each equation has been implemented in its own subsystem to make it easy to obtain a clear view of the model structure. These subsystems can be seen in Figure 2.7-2.9.

In addition a Speed block is added to transform the input to a vehicle speed. If a speed controller is to be designed, a accurate model should be implemented in this block. Currently it only contains a simple first order differential equation to create dynamic response to input change. This way the vehicle accelerates until the speed has converged to the value given by the throttle input.

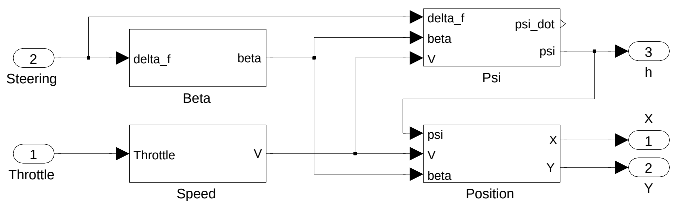


Figure 2.6: The Kinematic Single Track Model implemented in SIMULINK

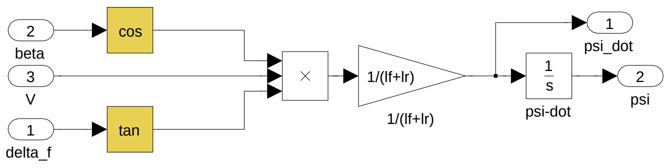


Figure 2.7: The psi SIMULINK block used in the Local Bug simulator

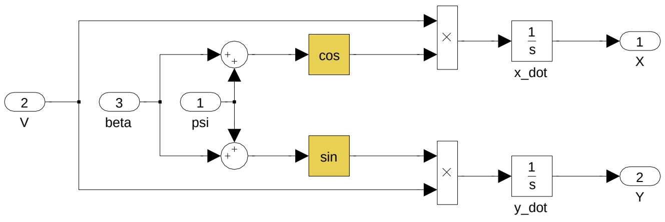


Figure 2.8: The position SIMULINK block

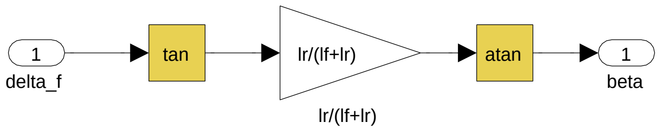


Figure 2.9: The beta SIMULINK block

2.5.2 Using the Simulator

The simulator consists of a SIMULINK model containing the vehicle model and a controller block. A simulation is executed by running the *run.m* file. This file contains vehicle initialization, waypoint selection and plotting functionality. By editing this *run* file it is possible to change model parameters and define which waypoints are to form the desired path. It is also possible to switch between rigid body model and kinematic model, but due to inaccurate tyre models and coefficients the rigid body model cannot be used. However, the simulator design makes it possible to modify or change the vehicle model used. The model outputs are transformed into the frame provided by measurement units to replicate actual position and orientation measurements. Real measurements have somewhat variable accuracy, while the simulated measurements are 100 percent accurate. Figure 2.10 gives an overview of the simulator implemented in SIMULINK. The simulator is included on the CD in Appendix C.

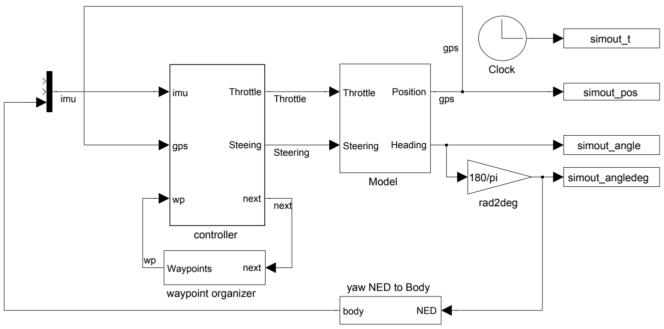


Figure 2.10: The Local Bug Simulator implemented in SIMULINK

3 Local Bug Test Platform

This chapter presents the different hardware components used on the Local Bug platform and the low level software used to communicate with the different components. It also presents the physical configuration used on the Local Bug e.g. positioning of measurements units and which interfaces used to communicate with them. Additional pictures of the Local Bug with the electronics are included in Appendix B.

3.1 Components

In this section the hardware components used on the Local Bug platform is presented.

3.1.1 Savage Flux

The Local Bug platform is based on a Savage Flux HP R/C Car developed by HPi Racing. It is a electric vehicle built on the Savage X chassis with a Flux Tork 2200Kv motor and a Blur Electronic Speed Control(ESC)[1] with a integrated cooling fan, which allows the ESC to operate at its ideal temperature. The Vehicle can be powered with either NiMH Batteries from 6- to 8-cell size, or dual LiPo

[1]Electronic Speed Controller: An electronic device that takes the power from the battery pack and the signal from the receiver and measures a certain amount of power to the car's motor. Only used in electric R/C cars, boats and planes. HPi Racing Webpage (2010)

Batteries with 2S 7.4v or 3S 11.1v for more power. The motor is mounted low in the chassis to provide a low CG for better control and cornering abilities.

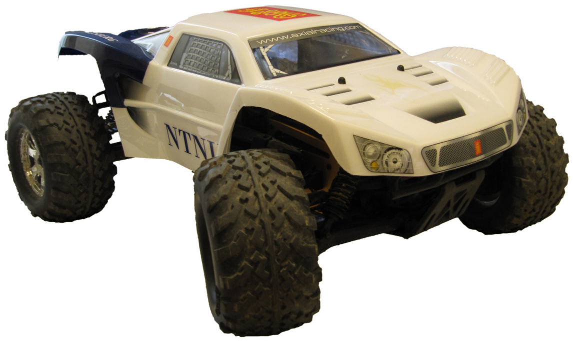


Figure 3.1: The Local Bug

Figure 3.1 is a picture of the Savage flux with the custom made Local Bug bodyshell.

3.1.2 Local Hawk Phoenix II

The Local Hawk Phoenix II is the new backbone of the Local Hawk hardware and was designed by Veierland (2010) to replace the first electronics. This time all of the control, communication and sensor interfaces are implemented on one circuit board. A Radiocrafts unit is included to make it possible to communicate with the unit through a wireless computer interface. The computational power is provided by two AVR XMEGA 256A3 MicroController Units(MCU), "MCU A" and "MCU B", with 256KB SRAM and a maximum clock frequency of 32MHz. MCU A is dedicated to handle all safety critical functionality such as parsing the sensor data, running control algorithms and controlling the actuators. MCU B handles the non critical tasks such as writing data to memory. It is also equipped with a multiplexer to control whether manual or autonomous control is to be sent to the actuators. This mechanism is implemented with a multiplexer where MCU B controls the selector signal. The two MCUs communicate through a Serial Peripheral Interface(SPI) Bus. Phoenix II has a Micro Secure Digital(SD) card socket connected to MCU B, making it possible to create data logs and store information which can be analysed when the vehicle is off line.

Each microcontroller has its own dedicated 10-pin 0.05 inch pitch connector through which their Programming and Debugging Interface (PDI) can be accessed. To program the microcontroller it is possible to use either a Joint Test Action Group (JTAG) device or an AVR In-System Programming (ISP) mkII programming device. A programming adapter is needed to connect the ISP's 6 pin connector to the 10 pin connector on the Phoenix II. Figure 3.3 shows the 10-pin female to 6-pin

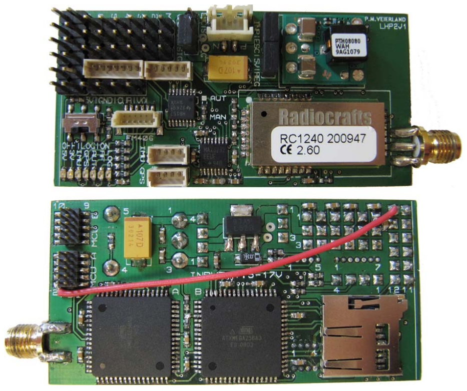


Figure 3.2: The Phoenix II Electronics

ISP and 10-pin JTAG adapter used to program the microcontrollers. The AVR ISP mkII device needs to be connected to the target power to interface the PDI. Since the controller algorithm runs on MCU A, this is the microcontroller which will be programmed most frequently. The red wire on Figure 3.2 connects the MCU A connector to the system power making it easy to connect the ISP mkII. To access MCU B a wire has to be connected temporarily between system power and pin 2 on the programming adapters JTAG connector(Veierland, 2010).

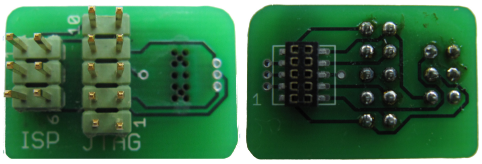


Figure 3.3: The programming adapter

The Local Hawk Phoenix II is mounted at the front of the Savage Flux inside the radio box. To make enough room, one of the plastic walls inside the radio box has been removed. As a side effect this creates an opening out of the box exposing the electronics. Although the opening is vertical, facing backwards and covered from above it is still vulnerable to rain and wet driving conditions. The Phoenix II is mounted by sliding the Radiocrafts antenna connector into one of the slots carved into the side compartment at the front of the radio box and attach the antenna on the outside of the box. Either of the slots can be used and the reason for having both is to be able to mount the Phoenix II in such a way that the desired side is easy accessible. If the log switch is going to be used it is convenient that it is facing upwards. On the other hand if the MCUs are going to be reprogrammed frequently it is desirable to have the 10-pin connectors facing up.

The Phoenix II has 6 servo output ports and can be powered through the servo 0 port. Connecting the ESC to servo port 0 while the jumper configuration is set to

ESC the speed is controlled through the same port as the power is supplied. This solution is used to power the card on the Local Bug test platform. The steering servo is connected to servo port 1, which both sends the control signal and supplies 5V to the servo.

3.1.3 GPS Receiver

The GPS module to be used is a Venus634FLPx 14 channel GPS receiver with a maximum update rate of 10Hz. It is $10mm \times 10mm \times 1.1mm$ in size and an accuracy of $2.5m$ CEP[2] according to the data sheet. It tests 8 million time-frequency hypothesis per second and has a open sky cold start of 29 seconds. One of the error sources for GPS called *multipath* is discussed by Vik (2009). This error arises when the satellite signal bounces form buildings or other objects before reaching the GPS module causing an increased signal propagation time. Since the GPS calculations are based on the transfer time between the satelite the receiver this introduces an error. The Venus634FLPx is equipped with multipath detection and suppression to minimize the errors caused by this phenomenon. The GPS module sends National Marine Electronics Association(NMEA) messages, in accordance with the VENUS634FLPx Data Sheet, through USART and is connected to Phoenix II 3.3V, RX, TX and GND ports displayed in the upper right corner of Figure 3.4.

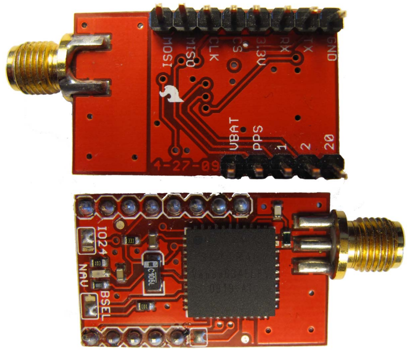


Figure 3.4: GPS receiver module

The GPS module is mounted in the vehicle in a similar manner as the Phoenix II. A hole in the radio box's rear compartment has been made to fit the antenna connector. When the antenna is fitted to the module through the hole the module is protected by the radio box while the tip of the antenna easily can be mounted on the vehicles roll bar. The top of the roll bar is the highest point on the savage chassis and is the most suitable place to have the GPS antenna.

[2]CEP: Circle of Equal Probability is defined as the radius of a circle, centred about a exact position, whose boundary is expected to include 50% of the measurements.

3.1.4 Inertial Measurement Unit

The MTi is a miniature, gyro-enhanced Attitude and Heading Reference System (AHRS). Its internal low-power signal processor provides drift-free 3D orientation as well as calibrated 3D acceleration, 3D rate of turn and 3D earth-magnetic field data.

Xsens User Manual

The Inertial Measurement Unit(IMU) is a XSENS MTi-28A53G35 commercial IMU module, the same unit that has been used in the previous Local Hawk Projects. It supports multiple ways of representing the output data, either as unit quaternions, rotation matrix or Euler angles. Since it is assumed that the car is driving with only planar motions the roll and pitch angles are not relevant. The singularity at 90 degrees pitch is omitted and there is no reason to include extra states by using rotation matrix or quaternion representation.

Figure 3.5: Xsens MTi

The IMU is powered at 5V and has an RS232 interface with a baud rate of 115200; 8 data bits, no parity and 2 stop bits. Xsens (2008) describes the messages used to configure and communicate with the MTi sensor. The IMU connector cable was disassembled in order to create a communication interface between the IMU and The Phoenix II. In accordance with Xsens (2006) the GND, VCC, RX and TX signal from the IMU soldered to a custom made cable with a MOLEX 4-pin female connector which fits the IMU input port on the Phoenix II.

Hagen et al. (2009) claims to have had problems with the consistency of the yaw measurement due to magnetic disturbances caused by engine power wires located close to the IMU. During the programming stage of this project it is also observed yaw drifting when the IMU is close to magnetic fields. In order to avoid such disturbances while on the Local Bug platform, the IMU is mounted as far from the electric motor as possible without exposing the unit to any higher risk if malfunctions would cause a crash. The IMU is mounted on the top lid of the radio box. A plastic heightening on the lid had to be removed in order to fit the IMU.

3.1.5 R/C Receiver

The R/C receiver used on the Local Bug is a Futaba R617FS 2.4GHz FASST 7-Channel Receiver (Figure 3.6). It is small, measuring only 41.6 × 27.5 × 9.2mm, and weighs only seven grams. It is powered by 5V and drains 80mA at no signal. Dual Antenna Diversity allows 2.4GHz FASST Futaba transmitter to select the best reception between the two receiver antennas with no signal loss.

Figure 3.6: Futaba Receiver

3.2 Drivers

This section presents the drivers and basic functions used to initialize and communicate with the hardware presented in the last section. In addition the basic software structure is presented in order to give an overview of the base design. Veierland (2010) has written a number of drivers during the development of the Phoenix electronics. Most of the drivers presented here are based on these beta drivers in order to stay consistent in the software development within the Local Hawk Project.

3.2.1 System Clock Initialization

To initialize the system clock to 32 MHz, the XMEGA Clock System driver provided by ATMEL is used. `sysclk_init()` is as follows:

```
CLKSYS_Enable( OSC_RC32MEN_bm );
do {} while ( CLKSYS_IsReady( OSC_RC32MRDY_bm ) == 0 );
CLKSYS_Main_ClockSource_Select( CLK_SCLKSEL_RC32M_gc );
```

It activates the 32MHz oscillator and waits until it is stable before selecting is as the system clock.

3.2.2 Servo Driver

Veierland (2010) wrote the servo driver and provided it with the Phoenix II in order to obtain easy servo initialization and use. Each servo output is configured in the servo header file `servo.h`. Defining SERVO_n_EN as 1 enables servo n and sets it up initialization. The SERVO_n_REVERSED defines if the servo direction should be reversed. Servos are controlled with Pulse Width Modulated (PWM) signals in a range from $1ms$ to $2ms$ as high period. The servo_init function utilizes the definitions made in the header file and initializes the servo outputs accordingly. To control the servo output the function SERVO_n_SETPOS(value) is used. It takes a byte and sets the PWM period. 255 gives a PWM period of $2ms$ while 0 gives $1ms$, if the SERVO_n_REVERSED to set to 0.

3.2.3 USART Driver

Universal Synchronous/Asynchronous Receiver/Transmitter (USART) communication is used for multiple purposes on the Local Bug. Both the GPS and the IMU uses it for both configuration and delivering measurement data. In addition USART communication has been used to write out messages to a computer for debugging purposes. The USART driver consists of `XUSART.c` and `XUSART.h` and provide functions for initialization and transmitting/receiving messages. XUSART_INIT(_X) initializes the USART according to the configuration set in `XConfig.h`. The configuration used for the debug communication is defined as follows:

```
#define XUSARTF0_BIND XUSART_DBG_
#define XUSART_DBG XUSARTF0
#define XUSART_DBG_BAUD 19200
#define XUSART_DBG_RXBUFSIZE 128
#define XUSART_DBG_TXBUFSIZE 64
#define XUSART_DBG_DREINTLVL USART_DREINTLVL_LO_gc
#define XUSART_DBG_RXCINTLVL USART_RXCINTLVL_LO_gc
```

PORTF is used for the debug USART, even tough this is designed as a GPS connection port and is connected to a level converter to be able to communicate with the LHP Venus GPS board(Veierland, 2010). This GPS board was not available during the development of the Local Bug Platform. Since USARTF0[3] is connected to a level converter it can communicate directly with a computer through the RS232 interface making this a easy way to send debug messages. The USART driver also uses `XBuffer.c` and `XBuffer.h` to create both transmit (TX) and receive (RX) buffers for the USART. In addition the USART driver provides two functions.

[3]USART interface 0 on port F(ATMEL, 2010)

```
bool XUSART_TXBuffer_PutByte( XUSART_t * xusart, uint8_t data );
void XUSART_TXBuffer_Transmit( XUSART_t * xusart );
```

These two functions are used to place a byte in the TX buffer and send it.

3.2.4 GPS Driver and Parser

As earlier mentioned the GPS module used on the Local Bug Platform is not supported by the two GPS connectors implemented on the Phoenix II. It is connected to the Berg Connectors connected to port E, pin 6 and 7 on MCU A. These pins can be used for USART messaging and is set up to be initialized as XUSARTE1 in `XConfig.h`.

```
#define XUSARTE1_BIND XUSART_GPS_
#define XUSART_GPS XUSARTE1
#define XUSART_GPS_BAUD 115200
#define XUSART_GPS_RXBUFSIZE 128
#define XUSART_GPS_TXBUFSIZE 64
#define XUSART_GPS_DREINTLVL USART_DREINTLVL_LO_gc
#define XUSART_GPS_RXCINTLVL USART_RXCINTLVL_LO_gc
```

The initialization of the GPS communication is done by using functions defined in the GPS driver(`GPS.c, GPS.h`).

```
XUSART_INIT( XUSART_GPS );
GPS_t gps;
GPS_Init( &gps, &XUSART_GPS.rxBuffer);
```

First the USART communication is initialized as XUSART_GPS, and then GPS_t is defined as the GPS parse structure and initialized through the GPS_Init function. This function also sets the RX buffer for the GPS USART as the message input buffer for the GPS parser.

The GPS Parser uses the `nmea.h` and `nmea.c` to obtain data stored in the RX buffer and stores it in the GPS_t structure's output variable. Since the GPS supports various data messages the GPS_Parse returns the message ID. This way the message can be recognised and data can be read from the message. However, since only the longitude and latitude data are used, the GPS are set up only to send one kind of message containing the wanted information. The GPS module can be configured by use of the GPS Viewer (2010) found on the Sparkfun webpage. This program can is also used to set the GPS measurement frequency, which is set default $1Hz$. To insure frequent GPS updates the frequency is $10Hz$ which is the maximum for this GPS module.

The NMEA parser defines structures compatible with the NMEA messages. When a message has been successfully parsed the output variable in the GPS_t structure can be redefined a NMEA message structure corresponding to the message id returned by the parser. The following code is used to obtain the GPS data.

```
uint8_t tmp;
NMEA_GPGGA_t package;
if((tmp=GPS_Parse( &gps ))!=0){
    if(tmp==NMEA_MESSAGE_TYPE_GPGGA){
        package = *(NMEA_GPGGA_t *)gps.output;
        double longitude=package.longitude;
        double latitude=package.latitude;
    }
}
```

If GPS_Parse returns a number different from zero, meaning a parse was successfully conducted, the message id is compared to the wanted message id. When the message id is equal to the wanted id the output data gets stored in the NMEA message structure *package*. Multiple message types can be supported by defining message structures for each type and store the data in the right structure based on the message id returned by the GPS_parse.

Transforming Ellipsoidal Coordinates to NED

The waypoints are defined in the NED frame and thus the GPS coordinates must be transformed into the same frame. This can transformation can be defined in two ways, with different accuracy and complexity. Drake (2002) defines a three step process converting longitude, latitude and height into the East-North-Up (ENU) frame. This process can also be used to transform the coordinates into the NED frame with just one modification. The three steps are:

1. Determine latitude, longitude and height of reference point.
2. Express small changes in latitude, longitude and height in ECEF coordinates.
3. By means of a rotation, displacements in ECEF coordinates are transformed to ENU coordinates.

In this thesis the rotation in the last step is changed in order to obtain the coordinates in the NED frame. The reference point can be chosen as the arbitrary point (l, μ, h) in order to derive the general equations. Longitude, latitude and height is transformed to Cartesian ECEF coordinates in accordance with Equation (3.1)-(3.3):

$$x_e = \left(\frac{a}{\chi} + h\right)\cos\mu\cos l \tag{3.1}$$

$$y_e = \left(\frac{a}{\chi} + h\right)\cos\mu\sin l \tag{3.2}$$

$$z_e = \left(\frac{a(1-e^2)}{\chi} + h\right)\sin\mu \tag{3.3}$$

where

$$\chi = \sqrt{1 - e^2 \sin^2 \mu} \tag{3.4}$$

a and e are the semi-major axis[4] and the first numerical eccentricity[5] of the Earth respectively. Small changes in latitude, longitude and height must be expressed in ECEF coordinates by Taylor-expanding Equation (3.1)-(3.1) about $\mu \to \mu + d\mu$, $l \to l + dl$ and $h \to h + dh$. This yields 3 quite comprehensive equations, which will not be presented here. They are however listed in Drake (2002). *The nth-Degree Taylor Polynomial* are presented in the *Preliminaries* of this thesis.

The rotation matrix from NED to ECEF is the product of two principal rotations: first a rotation l about the z-axis, and second a rotation $-\mu - \frac{\pi}{2}$ about the y-axis. Using trigonometric formulas: $\cos(-\mu - \frac{\pi}{2}) = -\sin\mu$, and $\sin(-\mu - \frac{\pi}{2}) = -\cos\mu$, yields(Fossen, 2002):

$$\begin{aligned}
\boldsymbol{R}_n^e &= \boldsymbol{R}_{z,l}\boldsymbol{R}_{y,-\mu-\frac{\pi}{2}} \\
&= \begin{bmatrix} \cos l & -\sin l & 0 \\ \sin l & \cos l & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} -\sin\mu & 0 & -\cos\mu \\ 0 & 1 & 0 \\ \cos\mu & 0 & -\sin\mu \end{bmatrix} \\
&= \begin{bmatrix} -\cos l \sin\mu & -\sin l & -\cos l \cos\mu \\ -\sin l \sin\mu & \cos l & -\sin l \cos\mu \\ \cos\mu & 0 & -\sin\mu \end{bmatrix}
\end{aligned} \tag{3.5}$$

Since ${\boldsymbol{R}_b^a}^\top = \boldsymbol{R}_a^b$ the rotation matrix from ECEF to NED is

$$\begin{bmatrix} x_n \\ y_n \\ z_n \end{bmatrix} = \begin{bmatrix} -\sin\mu\cos l & -\sin\mu\sin l & \cos\mu \\ -\sin l & \cos l & 0 \\ -\cos\mu\cos l & -\cos\mu\sin l & -\sin\mu \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \end{bmatrix} \tag{3.6}$$

The last stage in the process to convert ellipsoidal ECEF coordinates to NED is to rotate the equations obtained from the Taylor-expansion. This is done by substituting the Taylor-expanded equations into Equation (3.6) and neglecting terms of $\mathcal{O}(d\theta^3)$, $\mathcal{O}(dhd\theta^2)$ and higher, where θ is either μ or l. This yields the following

[4]The semi-major axis is half the longest diameter in an ellipse.

[5]The eccentricity of an ellipse is the ratio of the distance between the foci to the length of the major axis.

equations for deviation in the NED frame.

$$\begin{aligned}
dx_n &= \left(\frac{a(1-e^2)}{\chi^3} + h\right) d\mu + \frac{3}{2} a \cos\mu \sin\mu e^2 d\mu^2 + dh d\mu \\
&+ \frac{1}{2}\sin\mu\cos\mu\left(\frac{a}{\chi} + h\right) dl^2 \\
dy_n &= \left(\frac{a}{\chi} + h\right)\cos\mu dl - \left(\frac{a(1-e^2)}{\chi^3} + h\right)\sin\mu d\mu dl + \cos\mu dl dh \\
dz_n &= \frac{1}{2} a \left(1 - \frac{3}{2} e^2 \cos\mu + \frac{1}{2} e^2 + \frac{h}{a}\right) d\mu^2 - dh \\
&+ \frac{1}{2}\left(\frac{a\cos^2\mu}{\chi} + h\cos^2\mu\right) dl^2
\end{aligned} \tag{3.7}$$

These equations are quite complex and a less computational demanding transformation would be desirable on the 8-bit microcontroller.

Clynch (2006) presents an alternative approach to the transformation. This method is based on Geodecy[6] and utilizes a more intuitive approach when deriving the equations.

If the Earth was spherical the distance in *north* and *east* direction would be the same as calculating the arc length of two circles. The length of a circle arc is defined in Equation (6). Latitude is the angle in the *north* direction defined as zero at the equator and the radius of the circle would be the radius of the Earth. The same principle can be used in the *east* direction. However, the radius will not be equal to the Earth radius, since the size of the circle varies as a function of the latitude. *North* and *east* position as function of longitude, latitude and height are

$$\begin{aligned}
dx_n &= r d\mu \\
dy_n &= r \cos\mu dl
\end{aligned} \tag{3.8}$$

where r is the radius of the Earth or more precisely the distance between the circle arc and the centre of the Earth. When considering a ellipsoidal Earth the same equations can be applied. However, a new radius has to be defined for each of the directions. The radius used for longitude is denoted by r_n, where the subscript n stands for "normal". This is because the radius is defined as a line perpendicular to the ellipsoid surface originating at the chosen latitude and ends when it intersects the polar axis. r_n can be found as a function of the Earth equatorial radius (a), eccentricity (e) and the geodetic latitude (μ), see Equation (3.9).

$$r_n = \frac{a}{\sqrt{1 - e^2 \sin^2\mu}} \tag{3.9}$$

Geodetic latitude is defined as the angle between the r_n line and a horizontal line at the intersection point and must not be confused with geocentric latitude.

[6]Geodecy, also named geodetics is the scientific discipline that deals with the measurement and representation of the Earth, including its gravitational field, in a three-dimensional time-varying space.

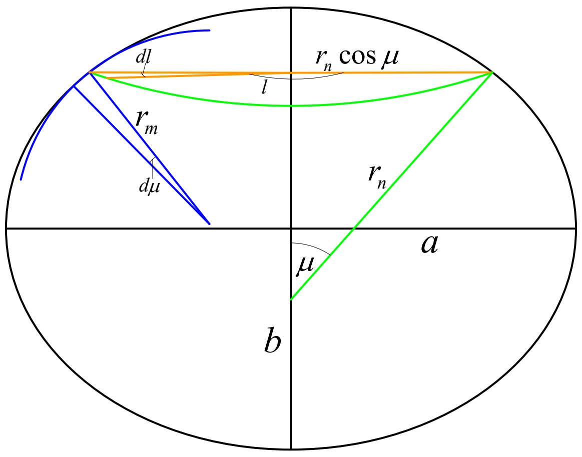


Figure 3.7: The Normal and the Meridian radii

The radius used for latitude is denoted r_m, where the subscript m refers to meridian which is the name of the lines running from north to south on a globe. r_m is the radius of a circle that is tangent to the ellipsoid at chosen latitude and has the same curvature as the ellipsoid in the north-south direction there. r_m can also be calculated from the Earth equatorial radius, its eccentricity and the latitude.

$$r_m = \frac{a(1-e^2)}{(1-e^2\sin^2\mu)^{\frac{3}{2}}} \tag{3.10}$$

The complite transfomation is

$$\begin{aligned} dx_n &= r_m d\mu \\ dy_n &= r_n \cos\mu dl \end{aligned} \tag{3.11}$$

where r_n and r_m are given by Equation (3.9) and (3.10). This only calculates positions on the surface of the Earth. When used with vehicles at various heights, the height variable should be added to the radius in order obtain the correct position. The geodetic latitude and both radii are illustrated in Figure 3.7. a and b denotes the Earth's semi-major and semi-minor axis respectively.

Comparing these to approaches gives a choice between easy computation and accuracy. The first method is more accurate and also more versatile. By neglecting terms, the accuracy and the computational power required diminishes. Notice that if all terms containing at least one product of the position deviations, that is $\mathcal{O}(d\theta^2)$,

$\mathcal{O}(dhd\theta)$ or higher, Equation (3.7) becomes

$$\begin{aligned} dx_n &= \left(\frac{a(1-e^2)}{(1-e^2\sin^2\mu)^{\frac{3}{2}}} + h\right) d\mu \\ dy_n &= \left(\frac{a}{\sqrt{1-e^2\sin^2\mu}} + h\right) \cos\mu dl \\ dz_n &= -dh \end{aligned} \tag{3.12}$$

This is the exact same equations as derived in the second method, but with the height added to the radius. In other words the first and second approach yields the same result. However, the neglected Taylor-expansion terms can be used to add extra accuracy when moving away from the reference point. Since the Local Bug platform only will move within reasonable local areas there is no need to use the extra computational power. For more information on these transformations and GPS theory in general, see Zhao (1997), Drake (2002), Clynch (2006) and Forssell (2003).

Equation (3.12) is implemented on the Local Bug platform. The code used is included below.

```
/*Calculated at initialization*/
double a = 6378137;// m (The Earth's semi-major axis)
double e = 0.08182;// (The Earth's eccentricity)
long double lat=origo.latitude*3.14/1800000000;
double h = origo.altitude/10;
double r_p = a*(1-pow(e,2)) / pow(1-pow(e,2)*pow(sin(lat),2), 3/2);
double r_e = a / pow(1-pow(e, 2*pow(sin(lat),2)), 1/2);
/*calculated for each iteration of the controller*/
double delta_longitude=(pakke.longitude-origo.longitude);
double delta_latitude=(pakke.latitude-origo.latitude);
delta_E = ((delta_longitude)*((r_e+h)*cos(lat)))*3.14/1800000000;
delta_N = ((delta_latitude)*(r_p+h))*3.14/1800000000;
```

The constants are calculated when the origin of the NED frame is determined, while the rest is calculated before a new iteration of the controller is initiated.

3.2.5 IMU Driver and Parser

The communication with the IMU is also done through USART. The `MTComm.c` provides three functions. MTComm_Initialize initializes a MTComm_t structure defined in `MTComm.h`. MTComm_Parse tries to parse a message from the USART RX buffer and returns the message id. Xsens (2008) defines the different messages used to configure and communicate with the Xsens MTi unit. MTComm_SendMessage takes a message id and the data to be transferred, arranges them correctly in the TX buffer and conducts a transmission.

`IMU.c` has been created in order to avoid having a long initialization routine in the beginning of the main file. The imu_init function runs the necessary USART initialization in accordance with `XConfig.h`.

```
#define XUSARTC0_BIND XUSART_IMU_
#define XUSART_IMU XUSARTC0
#define XUSART_IMU_BAUD 115200
#define XUSART_IMU_RXBUFSIZE 128
#define XUSART_IMU_TXBUFSIZE 64
#define XUSART_IMU_DREINTLVL USART_DREINTLVL_LO_gc
#define XUSART_IMU_RXCINTLVL USART_RXCINTLVL_LO_gc
```

It also calls the MTComm_Initialize function to complete the communication start up. Next it starts parsing the data in the RX buffer in order to check the state of the IMU. The MTi has two states, Configure and Measurement. At power up the IMU starts its wakeup routine and sends a wakeup message. If a "wakeup acknowledge" message is received within 500ms the device enters the configure state. Without an acknowledge response the device enters the measurement state.

After setting up the communication, the *imu_init* function waits for a message from the IMU. When a wakeup message is received, a "wakeup acknowledge" is sent to the device. After a delay a "goto config" message is sent to ensure that the IMU has entered the configure state. Then configuration messages are sent with the wanted output setting.

The IMU is configured to deliver calibrated orientation data as Euler angles. FIXED1220[7] is set as the output data format and measurements are sent at 100Hz. Since the IMU sends data at a constant rate unaffected by state of the MCU, a timer interrupt is set up to control when the parser is supposed to be executed. In order to keep the USART RX buffer from getting full causing loss of data, the parse routine must be called at least as frequently as data is delivered by the IMU. Several timer periods has been tested and it seems that anything under 5000Hz causes the IMU communication to freeze. The timer interrupt is set up as follows:

```
/*set the timer period*/
TC_SetPeriod( &TCC0, 0x1338 );
/*select event channel and config input capture*/
EVSYS.CH1MUX=EVSYS_CHMUX_TCC0_OVF_gc;
TC0_ConfigInputCapture( &TCC0, TC_EVSEL_CH1_gc );
/*set interrupt level*/
TC0_SetOverflowIntLevel( &TCC0, TC_OVFINTLVL_LO_gc );

/* Select clock source to start the timer */
TC0_ConfigClockSource( &TCC0, TC_CLKSEL_DIV1_gc );
```

[7]FIXED1220: Fixed point signed 12.20 is a 32 bit fixed decimal number format. The 12 first bits are the main number while the 20 last represents the decimals. Negative numbers are represented as two's compliment.

All tests have successfully been conducted without any problems with these settings. The routine is called at frequency of:

$$\frac{32MHz}{4920} \approx 6500Hz \tag{3.13}$$

The timer Interrupt Service Routine (ISR) calls the parse routine. If a message id different from 0 is returned it transforms the data into a more usable data format and stores it in the variables used by the controller. This makes the yaw data variable updated as soon as possible after an IMU message is received.

```
uint8_t tmp;

tmp = MTComm_Parse( &imu );

if(tmp!=0){
    xsens_data * imu_data = imu.data + 2;
    data->timestamp = imu_data->timestamp;
    /*Roll*/
    if((int32_t)imu_data->euler_roll≥0){
        data->euler_roll =((imu_data->euler_roll)>>20);
    }else{
        data->euler_roll =-~((~(imu_data->euler_roll))>>20);
        data->euler_roll=360-data->euler_roll;
    }
    /*Pitch*/
    if((int32_t)imu_data->euler_pitch≥0){
        data->euler_pitch =((imu_data->euler_pitch)>>20);
    }else{
        data->euler_pitch =-~((~(imu_data->euler_pitch))>>20);
        data->euler_pitch=360-data->euler_pitch;
    }
    /*Yaw*/
    if((int32_t)imu_data->euler_yaw≥0){
        data->euler_yaw =((imu_data->euler_yaw)>>20);
    }else{
        data->euler_yaw =-~((~(imu_data->euler_yaw))>>20);
        data->euler_yaw=360-data->euler_yaw;
    }
}
```

First the 32bit measurement is bit shifted 20 places to the right, removing the 20 decimal bits. Then the two's complement numbers are converted to sign bit format and stored in the data variable. This procedure gives one degree accuracy, if more decimals are desired the conversion has to be modified. However, for this project one degree precision is satisfactory.

3.2.6 SPI On-board Communication

The on-board SPI communication is an important part of the system. Since MCU B handles external communication and saving log data to memory, the two MCUs has to be able to transfer data between them. If for example, waypoints are sent to the vehicle via the radio link MCU B has to be able to forward this information to MCU A. Veierland (2010) developed the SPI driver as a part of the Phoenix II design. Currently the SPI communication is only used to transfer measurement data from MCU A to MCU B in order to create a log file on a Micro SD card.

`XSPI.c` provides functions for initialization and communication for both master and slave units. MCU A is configured as the SPI master while MCU B is the slave. Both initialize functions creates RX buffers and connects the SPI driver to a port.

3.2.7 MicroSD Driver and Logging Routine

Phoenix II uses SPI to interface the microSD's internal storage controller(SD Group, 2006). For this SPI communication MCU B is initialized as the master. Veierland (2010) chose to use FAT32 file format on the SD card, making it possible to insert the card into a computer and get access to the data without any custom software. The FAT32 library used is the portable "FatFS" implementation developed by Elm Chan (2010). It can be used on various microcontrollers and does not require any specific memory type since it is fully separated from the the disk I/O layer. However, some modifications are necessary to make it work on Phoenix II. Chris H (2010) provides a guide in how to port the FatFS library to work on XMega controllers. A few code lines had to be added to the "FatFS" source code and a timer interrupt had to be set up.

> *The diskio.c file communicates with the SD card and if something fails then it needs to know how long it has been waiting for a response from the card. If the card doesn't respond in a few milliseconds then the function fails. The diskio.c file has a function called void disk_timerproc() and it has to be called every 10ms.*
>
> Chris H (2010)

The logging is currently timed by MCU A, it sends a log data message via SPI to MCU B 6 times per second. This message contains the position in the NED frame and the yaw angle. MCU B is consistently checking if there is a log message in the SPI RX buffer. As soon as a complete message is detected MCU B checks if the log switch is active and if a log file is already open, before writing the information to the file. If the log switch is active, but no log file is open, MCU B mounts the MicroSD card, creates a log file and opens it. When the log switch is found to be inactive and a log file is open, MCU B closes the log file and unmounts the MicroSD card. The functions used to perform operations on the storage media is provided by the "FatFS" library and is described below.

```
/*Mount device*/
f_mount(0, &myfat);
/*Open/create file*/
f_open(&fil_obj, "test.txt", FA_WRITE | FA_OPEN_ALWAYS);
```

```
/*Write to file*/
f_printf(&fil_obj, " %d\n",(int)incomingMessage.north);
/*Close file*/
f_close(&fil_obj);
/*Unmount device*/
f_mount( 0, 0);
```

f_mount's first argument is the device number to be associated with this drive, in this case 0. The second argument is the work area/file system object to be used with this device. *f_open* creates a file object structure and opens "test.txt". The last argument specifies the type of access and open method to use on the file, the possible mode flags are listed in Table 3.1. By default the opening method is FA_OPEN_EXISTING. To unmount the device, use the *f_mount* function without sending any file system object. The complete logging routine on MCU B is added in Appendix A.2.

FA_READ	Data can be read from the file
FA_WRITE	Data can be written to the file
FA_OPEN_EXISTING	Opens the file. Fails if the file does not exist.
FA_OPEN_ALWAYS	Opens the file if it exists. If not a new file is created
FA_CREATE_NEW	Creates a new file. Fails if the file already exists.
FA_CREATE_ALWAYS	Creates a new file. If the file exists it is overwritten.

Table 3.1: f_open Mode Flags

3.2.8 PWM Detection

One channel on the receiver is used to switch between autonomous and manual control. Since the receiver only can output PWM signals, MCU B has to be able to detect changes in the PWM period. This can be done by starting a timer at the rising edge and reading the counter value when the falling edge occurs. The following code initializes the timer and configures it to throw interrupts on both rising and falling edges.

```
 /*set PE0 as input triggered on both edges */
    PORTA.PIN6CTRL = PORT_ISC_BOTHEDGES_gc;
    PORTA.DIRCLR = PIN6_bm;
    /*set PA6 as input to event channel 0 */
    EVSYS.CH0MUX = EVSYS_CHMUX_PORTA_PIN6_gc;
    /*enable input capture and select clock*/
    TC0_ConfigInputCapture( &TCE0, TC_EVSEL_CH0_gc );
    TC0_EnableCCChannels( &TCE0, TC0_CCAEN_bm );
    TC_SetPeriod( &TCE0, 0x7FFF );
    TC0_ConfigClockSource( &TCE0, TC_CLKSEL_DIV64_gc );

    /*Set interrupt level*/
    TC0_SetCCAIntLevel( &TCE0, TC_CCAINTLVL_LO_gc );
    PMIC.CTRL |= PMIC_LOLVLEN_bm;
```

The ISR detects if it was a rising or a falling edge that caused the interrupt. If it was a rising edge it resets the timer and exits the service routine. When a falling edge triggered the interrupt, the counter value is stored and compared with a value between the ones representing on and off switch values.

```
ISR(TCE0_CCA_vect)
{
    static uint16_t highPeriod;

    uint16_t thisCapture = TC_GetCaptureA( &TCE0 );

     /* if rising edge reset counter*/
    if ( thisCapture & 0x8000 ) {
        TC_Restart( &TCE0 );
    }
     /*else save high period*/
    else {
        highPeriod = thisCapture;

                /*set LED and mux based on the high period*/
        if(highPeriod>749){
            PORTF.OUT|=0x04;
            PORTF.OUT&=~0x10;
        }else{
            PORTF.OUT&=~0x04;
            PORTF.OUT|=0x10;
        }
    }
}
```

4 Controller Software Implementation

In addition to the framework presented in Chapter 3 there is a obvious need for a control system. A controller monitors and affects the operational conditions of a given dynamical system. The controller on the Local Bug is going to include two separate segments. The first is the guidance controller. It uses position measurements and waypoint data to compute a heading which if followed will lead the Local Bug to the next waypoint. The second segment is the heading controller which monitors the vehicles heading and uses the steering angle of the wheels to maintain the desired heading.

This chapter introduces the heading controller and the guidance schemes for the Local Bug vehicle. SIMULINK implementations are presented and Real-Time Workshop is used to generate C-code so the controller can be implemented and tested on the Local Bug Platform.

4.1 Heading Control

Because the Local Bug is not going to be exposed to disturbances of significant proportions, it does not require a advanced heading controller. Utilization of the steering abilities comes as one of the most important requirements.

The controller implemented on the Local Bug is a Proportional(P)-controller. This is one of the most intuitive and basic controllers. The idea behind a P-controller is to use the difference between the desired value and the measured value multiplied with a controller gain as the input to the system. The higher the gain the more aggressive the controller becomes.

The Local Bug's maximum steering angle is about 30° and as earlier pointed out it is important that this is utilized. With the gain set to one, the steering angle will be equal to the error. This will turn the wheels to point in the direction of the desired heading unless the error is greater than 30°. If it is the steering angle will be at its limit turning the car as quickly as possible. This seems like reasonable controller behaviour, but is adjusted to work satisfactory during the testing presented in Chapter 5.

There is however one problem with just applying a P-controller for the heading control. If the desired heading is 350° and the current heading is five degrees, giving an error of 345°, causing the vehicle to turn right. This is however a very inefficient way to achieve the desired heading, since it actually is only ten degrees from the desired heading. To avoid this, the following embedded Matlab function is used to calculate the actual error.

```
function e = fcn(e_in)
e = e_in;
if(norm(e_in)>180)
   e=e_in*(1-360/norm(e_in));
end
```

Figure 4.1 shows the heading controller implemented in SIMULINK.

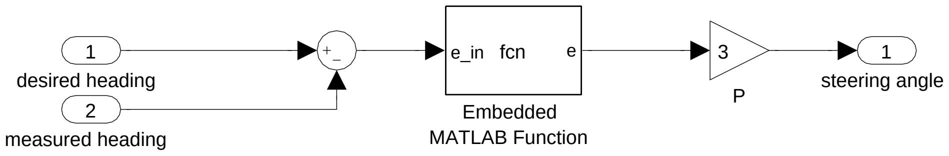


Figure 4.1: The Local Bug Heading Controller

4.2 Guidance

Vold (2009) discusses the use of guidance schemes in the Local Hawk Project. Three different algorithms were implemented in SIMULINK and tested with satisfactory results. The Line of Sight (LOS) and the Cross-Track Error (CTE) schemes all led to successful simulations guiding the vehicle through all waypoints. However, the cross-track error scheme with straight lines and circles introduced more complex equations without any increased accuracy. Because of this result the circles and lines approach will not be included in this thesis. The algorithms presented in this section is implemented in SIMULINK and included in the Local Bug simulator in Appendix C.

4.2.1 Line of Sight

The LOS algorithm is widely used method as it is both simple and efficient. It can be defined in two ways. The desired heading is either the direction to the next waypoint, or to a intersection point on the line between the last and the next waypoint somewhere ahead of the vehicle. The vector from the vehicle to the intersection point is called the LOS vector, while the LOS angle is defined as the angle of the LOS vector relative the inertial frame x-axis. Figure 4.2 illustrates the LOS vector and the desired angle χ. There are only two equations necessary when using the LOS scheme with the interaction point at the next waypoint. The first one is to compute the LOS angle as

$$\chi = atan2(y_{wp} - y(t), x_{wp} - x(t)) \tag{4.1}$$

where y_{wp} and x_{wp} is the coordinates of the next waypoint, while x and y are the current position of the vehicle. The $atan2(y, x)$ function is a four quadrant version of the $atan(\frac{y}{x})$ function. Resulting in an angle represented in the area $[-\pi, \pi]$ instead of $[-\frac{\pi}{2}, \frac{\pi}{2}]$. The second equation necessary is used to check if the vehicle is inside the circle of acceptance. This is done by checking if

$$[x_k - x(t)]^2 + [y_k - y(t)]^2 \leq R_{dist}^2 \tag{4.2}$$

where R_{dist} is the radius of the circle of acceptance.

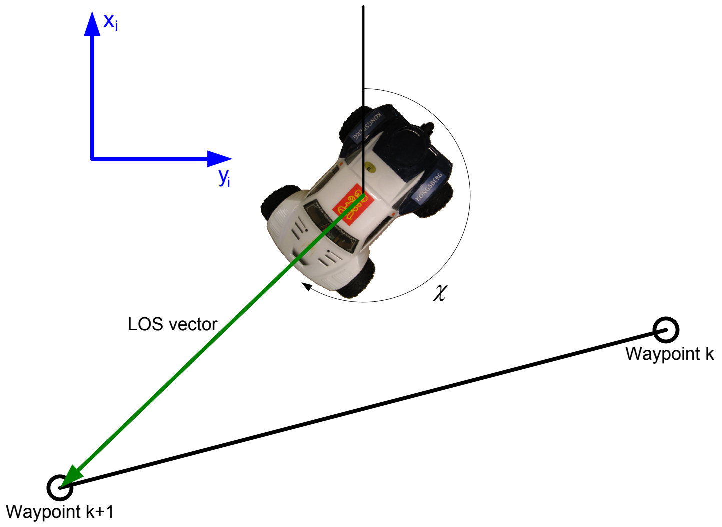


Figure 4.2: The Line of Sight approach. i is the inertial frame and χ is the desired heading.

There are 2 ways of implementing the algorithm in SIMULINK without loosing the possibility to generate C-code from it. It can either be implemented as a embedded MATLAB block or it can be implemented using SIMULINK blocks.

The latter possibility can be quite cumbersome since block diagrams can get complicated when using only the standard SIMULINK blocks. When using the embedded MATLAB block approach the algorithms can be implemented using standard MATLAB code. Since algorithms more often than not is presented as a number of equations, this most practical way of implementing algorithms. The LOS algorithm is implemented in a embedded MATLAB block as

```
function [ref_heading, thrust, next] =  los(x, y, wpn, wpl)

%The LOS angle
ref_heading=atan2((wpn(2)-y),(wpn(1)-x))*180/pi;

%The distance between the vehicle and the next waypoint
dist=sqrt((wpn(1)-x)^2+(wpn(2)-y)^2);

%Check whether the vehicle is within the circle of acceptance
if dist≤wpn(3)
     next=1;
else
     next=0;
end
%set the desired velocity
thrust = 1;
end
```

The *next* variable is used to signal that the controller is ready for the next waypoint. Currently the guidance block only sets a constant desired speed, but it is possible to implement other methods. If for example the vehicle is tracking straight lines at full speed it would have to lower the speed when approaching a waypoint in order to avoid flipping over.

4.2.2 Cross-Track Error

The cross-track error is the shortest distance between the vehicle and the desired path. More precisely it is the length of a vector that originates at the path ends at the vehicle while also being perpendicular to the path. It is desirable to minimize this error in order to follow the path as tightly as possible. Hence, the desired heading is a function of the cross-track error. This approach can be used with a variety of different path structures. As mentioned earlier Vold (2009) implemented and tested this approach with circles and lines without any significant improvement of the path following. This is why the algorithm implemented on the Local Bug platform only uses straight lines between waypoints as the desired path.

Figure 4.3 illustrates the CTE approach and the different variables involved. The desired heading angle is composed of two angles. One of them are the actual direction of the path to be followed and is denoted as χ_p. The second component is the angle of the desired velocity vector relative to the straight line between the

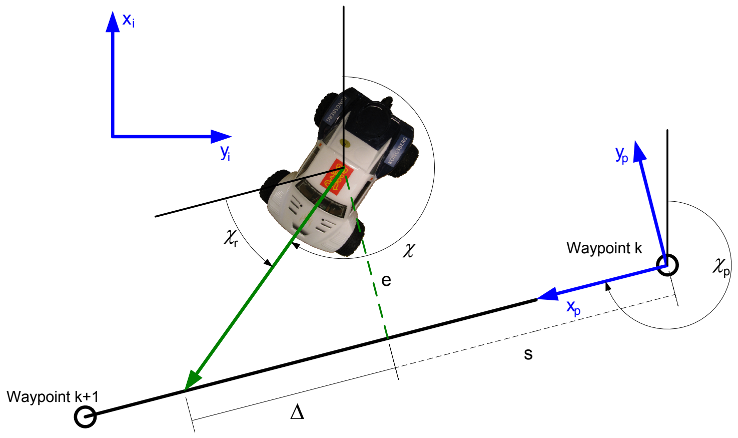


Figure 4.3: The Cross-Track Error Approach. i is the inertial frame and p is a frame reference frame parallel to the path. χ_p is the angle denoting the direction of the path to follow and χ_r angle of the desired heading relative the path. Δ and s are the look ahead and along path distance respectively while e is the cross-track error.

last waypoint and the next. This angle is denoted $\chi_r(e)$ as it is a function of the cross-track error e.

χ_p can be found by calculating the difference between the last and the next waypoint in x_i and y_i direction, and find the angle by using the $atan2()$ function. This is done in Equation (4.3).

$$\chi_p = atan2(y_{k+1} - y_k, x_{k+1} - x_k) \tag{4.3}$$

The desired direction of the velocity vector is also found from a geometric point of view. As it is supposed to minimize the cross-track error the fastest solution would be to go directly against the desired path. However, this approach ignores the fact that the main goal is to follow the path. To combine the goals of following the track while minimizing the cross-track error, the look-ahead distance Δ is introduced. This is how far in front of the current along-path position the vehicle should aim. The look-ahead distance is a tuning variable for the guidance controller, shorter the distance gives a more aggressive the controller is to minimize the cross-track error. $\chi_r(e)$ can be found as:

$$\chi_r(e) = atan2(-e, \Delta) \tag{4.4}$$

The cross-track error can be found rotating the vehicles position relative the last waypoint into a path-fixed frame:

$$\boldsymbol{\epsilon}(t) = \mathbf{R}(\chi_p)(\boldsymbol{p}(t) - \boldsymbol{p}_k) \tag{4.5}$$

where $\mathbf{R}(\chi_p)$ is the rotation matrix

$$\mathbf{R}(\chi_p) = \begin{bmatrix} \cos\chi_r & -\sin\chi_r \\ \sin\chi_r & cos\chi_r \end{bmatrix} \tag{4.6}$$

ϵ is a vector containing the cross-track error e and the along-path distance the vehicle has travelled since the last waypoint. This variable, denoted by s, can be used instead of the circle of acceptance to decide whether or not to change to the next waypoint. The total desired heading is then found by adding the to components together, as done in Equation (4.7).

$$\chi(e) = \chi_p + \chi_r(e) \tag{4.7}$$

The look-ahead distance cross-track error algorithm has been implemented in a embedded MATLAB block as follows:

```
function [ref_heading, thrust, next] = look_ahead(x, y, wpn, wpl)

persistent xl;
persistent yl;
persistent alpha;
persistent R;
    if isempty(xl)
        xl=wpn(1);
        yl=wpn(2);
        alpha = atan2(wpn(2)-wpl(2),wpn(1)-wpl(1));
    % The Rotation matrix
        R=[cos(alpha) -sin(alpha); sin(alpha) cos(alpha)];
    end
if (xl≠wpn(1))||(yl≠wpn(2))
    %The angle between initial reference frame and desired path
    alpha = atan2(wpn(2)-wpl(2),wpn(1)-wpl(1));
    % The Rotation matrix
    R=[cos(alpha) -sin(alpha); sin(alpha) cos(alpha)];
end
xl=wpn(1);
yl=wpn(2);

Delta = 2;    %The lookahead distance

%Vector consisting of along-path distance between vehicle and waypoint,
%and cross-path error
epsilon=R'*([x y]'-[wpl(1) wpl(2)]');

%Computing the desired heading angle
Khi_p=alpha;
Khi_r=atan2(-epsilon(2), Delta);
ref_temp=Khi_p+Khi_r;
ref_heading=ref_temp*180/pi;
thrust = 1;

%Check whether to change to next waypoint or keep current waypoint
    if norm([wpl(1) wpl(2)]-[wpn(1) wpn(2)])-epsilon(1)<wpn(3)
        next=1;
    else
        next=0;
    end
end
```

The waypoint switching condition uses the along-path distance to find out if the vehicle is close enough to the waypoint. Notice that does not consider the cross-track error at all. In other words the waypoint switch takes happens as long as the vehicle has moved far enough along the path. The switching criteria can be written as

$$(s_{tot} - s) \leq R_{dist} \tag{4.8}$$

where s_{tot} is the total along-path distance between the waypoints and R_{dist} is the along-path distance ahead of the waypoint where the vehicle should start following the next line.

4.3 The Local Bug Controller Block

The Local Bug controller Block contains the guidance scheme, all the controllers used on the system and some converter blocks. When C-code is generated, all functionality within this block will become a part of the C-function. Figure 4.4 shows the complete controller structure in the SIMULINK controller block. The Servo converter blocks are included in order to convert the controller output to values usable in the Phoenix II framework presented in Chapter 3. For steering, the output is converted from the area $[-30°, 30°]$ to the servo input area $[0, 255]$.

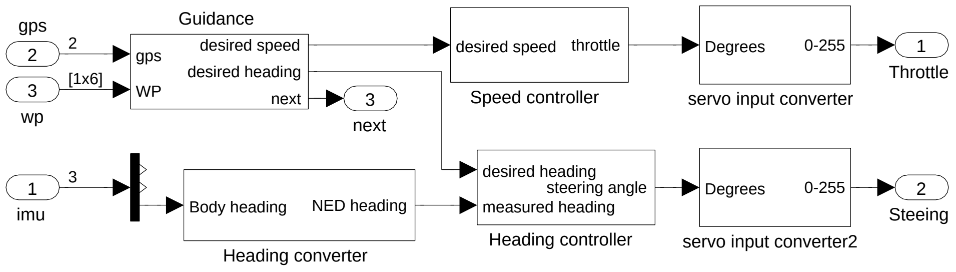


Figure 4.4: The Local Bug Controller Structure

Because the heading data from the IMU is represented as positive rotation about the body z-axis, while the guidance scheme defines heading as rotation about the NED z-axis a heading converter is also required. The IMU input contains roll, pitch and yaw. Roll and pitch are not used in any of the currently used controllers, but the framework is prepared for controller development. The waypoint signal contains two waypoints, last and next, with three values each. In addition to the desired position in the NED frame, each waypoint is also defined by the radius of the circle of acceptance. The GPS coordinates were transformed to the NED in the framework(Section 3.2.4) before sent to the controller. These values are therefore position in North and East given in metres. The next variable is sent to the framework to signal that the waypoint pointer should be incremented.

The speed controller is currently not doing anything but was added to make the control structure complete. Roll and Pitch from the IMU signal could be sent to this block to control speed based on indications of flipping over in any direction.

4.4 Real-Time Workshop

Mathworks Real-Time Workshop is a way to transform SIMULINK models into code which can run on other platforms than SIMULINK and MATLAB. It can build programs that can run in a variety of environments, including real-time systems and stand-alone simulations. Real-Time Workshop makes it possible to run SIMULINK models in real-time on a remote processor. High-speed stand-alone simulations can run on your host machine or on an external computer. It is also possible to use the Real-Time Workshop to implement hardware-in-the-loop simulations.

The Real-Time Workshop software includes an automatic C language code generator for SIMULINK. It generates C-code directly from SIMULINK block diagrams. This makes it possible to use SIMULINK to develop controllers for use in embedded systems.

One of the reasons for using Real-Time Workshop generated code running on the Local Beast is the improved user-friendliness. It is more intuitive to implement a controller as a SIMULINK model, than writing them as C-code. In addition, new controllers can be implemented without having to read up on the drivers for the different actuators and sensors. This is an advantage when the project equipment and code are passed on to new students and other projects in control design.

4.4.1 How Real-Time Workshop generated code works

The generated code has a standard structure with functions which control the model. To be able to use generated code it is important to know what tasks these functions perform. A brief description of the code structure is given below, for more details see the Real-Time Workshop 7, User's Guide. Since the code generated will be running on an AVR microcontroller it is reasonable use discrete methods for the controller to ease the computation load created by the Real-Time Workshop generated methods.

MdlInitializeSizes():

As described in the function name this method initializes the model sizes. Among these are the number of continuous states, the number of model outputs and inputs. This function should be called as a part of the system initialization.

MdlUpdate():

MdlUpdate is used to update the discrete states or similar type objects in all of the blocks within the model. If the SIMULINK model contains discrete integration this function must be called with constant intervals given by the step size used in the model.

MdlOutputs():

This function calls all the blocks in the model and makes them produce their output. In other words this function runs most of the calculations in the model and must be called on a regular basis to produce the servo settings needed to control the vehicle.

MdlTerminate():

Terminates the program if it is designed to run for a finite time. It destroys the real-time model data structure, deallocates memory, and can write data to a file. This function is obviously not relevant in this control application since controller will be sett up to run infinitely.

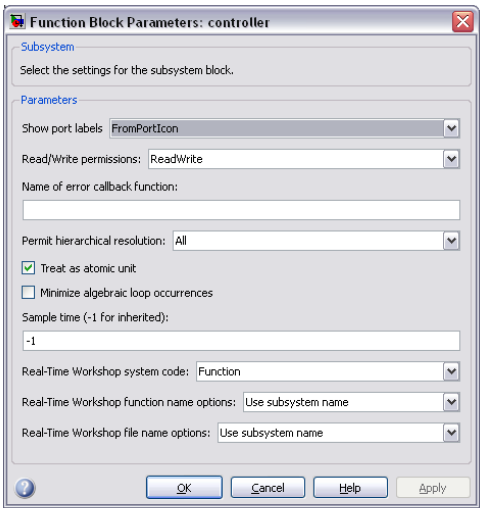
4.4.2 Code Generation

Skandan (2005b) and Skandan (2005a) provide step by step guides on how to use Real-Time Workshop to export C-code and use it in an application. The guides were more used for inspiration than as guide because they where for an older version of RTW and did not exactly fit the set up needed on the Local Bug Platform.

Preparing the SIMULINK Model

Skandan (2005a) recommend to use the discrete fixed-step solver when generating the C-code. To do this the controller will have to be moved to its own SIMULINK model because the vehicle model contains continues states which cannot be simulated with the discrete solver. However, the Local Bug Platform has been tested using code generated with the fixed-step ode5 solver without any problems at all. This is probably because the controller block does not contain any continues states and hence the integration methods are avoided. The safest approach is to follow the recommendation of Skandan (2005a), and relocate the controller before performing the code generation.

Before generating the C-code it is wise to assign names to the input and output signals of the SIMULINK block you want to export, in this case the controller block. This is done by right-clicking on the signal line, choosing *signal properties* and write the name in the signal name field. The variables used in the C-code will adapt this


Function Block Parameters: controller

Subsystem

Select the settings for the subsystem block.

Parameters

Show port labels: FromPortIcon

Read/Write permissions: ReadWrite

Name of error callback function:

Permit hierarchical resolution: All

☑ Treat as atomic unit

☐ Minimize algebraic loop occurrences

Sample time (-1 for inherited): -1

Real-Time Workshop system code: Function

Real-Time Workshop function name options: Use subsystem name

Real-Time Workshop file name options: Use subsystem name

Figure 4.5: The Subsystem Parameters Window

name, making the signal handling in c quite intuitive. It is also important to set the block up as an atomic unit, by right-clicking on the block and choosing *subsystem parameters*. Select the *Treat as atomic unit* box and set the Real-Time Workshop parameters as is Figure 4.5 and click apply. The subsystem block should now have a thicker edge to indicate that it is an atomic unit. This makes the RTW code provide the block as a C-file with the name of the subsystem block.

Real-Time Workshop Settings

The RTW settings is found under *Tools* ⇒ *Real-Time Workshop* ⇒ *Options*. Under *Hardware implementation* on the left menu, select Atmel as the *Device vendor* and AVR as the *Device type* to let RTW customize the C-code accordingly.

In the Real-Time Workshop menu the *System target file* is set to *grt.tlc* which is Generic Real-Time Target. C-code generated with the compiler optimization both switched on and off has been used on the vehicle without any visible differences, but it cannot be bad to use C-code optimized for faster runs. It is also important to deselect the *Generate makefile* box, to make it possible to relocate the generated code with ease. This topic will be covered later.

The model should now be ready for code generation. Right click on the controller block and select *Real-Time Workshop* ⇒ *Build Subsystem*. A window named *Build code for Subsystem:controller* should now appear. Click build in order to start the code generation. If the window closes without any error messages the generated code will be located in a folder named *contoller_grt_rtw*.

Relocate the Generated Code

The code generated by Real-Time Workshop uses a variety of header files form various locations in the MATLAB program folder. Locating all these dependencies and copy them manually would be quite cumbersome, luckily RTW provides a function designed for relocation purposes. *packNGo* uses the build info file and packages the model code, including all dependencies, in a zip file ready for relocation. Along with the C-code RTW creates a *buildInfo.mat* file. This is placed in the *controller_grt_rtw* folder. Load this file and use it to run *packNGo* to package the controller for relocation. This is done by typing the following commands in the MATLAB window.

```
load buildInfo.mat
buildInfo.packNGo
```

If the code is generated without disabling the *create makefile* option a error message will now appear. This is because the *packNGo* function does not support RTW generated makefiles. Disable the option and regenerate the code to be able to continue. When successfully completed, *controller.zip* will be created in the root folder of the SIMULINK model. This file contains all the files needed to successfully compile the program. Move it to the wanted location, extract and place all the files on the same level in a folder.

4.4.3 Combining the RTW Generated with the Framework

When all the files in the *zip* file has been extracted and placed in a folder with the Local Bug framework, it is time to combine the functionality in AVR Studio[1]. Create a project containing the Local Bug framework and add all the files from the controller zip file. The project settings has to be set correctly with the XMEGA256A3 as the device and $32MHz$ as frequency. Before *controller.h* can be included in the Local Bug main file it is necessary to state the program to be real-time executable. This is done by inserting a define at the top of the main file and any other files which includes *contoller.h*. Without this define, AVR Studio will fail to compile the program. The controller inclusion is done as follows:

```
#define RT

#include "controller.h"

extern void MdlInitializeSizes();
extern void MdlInitializeSampleTimes();
extern void MdlStart();
extern void MdlUpdate();
```

[1]AVR Studio 4 is the Integrated Development Environment (IDE) for developing 8-bit AVR applications in Windows NT/2000/XP/Vista/7 environments

```
extern void MdlOutputs();
extern void MdlTerminate();
extern ExternalInputs_controller controller_U;
extern ExternalOutputs_controller controller_Y;
```

The RTW standard methods need to be defined as external methods if they are to be called by the main file. In addition the input and output structures must also be included as external units in order to be able to access them. Inputs and outputs are found in the *controller_U* and the *controller_Y* structures respectively. *controller.h* defines these structures as

```
typedef struct {
  real_T imu[3];
  real_T gps[2];
  real_T wp[6];
} ExternalInputs_controller;

typedef struct {
  real_T Throttle;
  real_T Steering;
  real_T next;
} ExternalOutputs_controller;
```

The input and output structures each have three variables. These variables have adapted the signal names applied to the SIMULINK model before the code was generated. *imu*, *gps* and *wp* are inputs, while *throttle*, *steering* and *next* are outputs. The controller can now be used by following these steps:

1. Store measurements and waypoint data in the input variables.
2. Call the *MdlOutputs* function
3. Copy the output variables from the *controller_Y* structure.

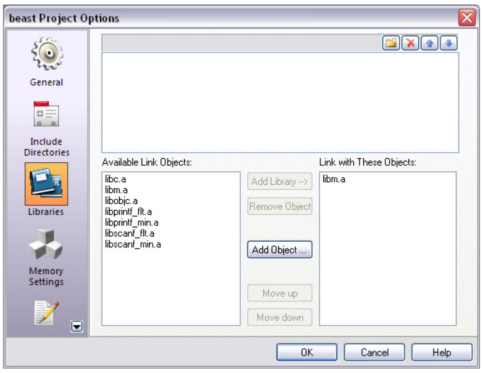
Some of the mathematical functions used by the controller require an additional library. Select *Project⇒Configuration Options⇒Libraries* and add *libm.a* to the list objects to be linked to the project, as illustrated in Figure 4.6.

AVR Studio should now be able to compile the source code.

4.5 The Local Bug Program Structure

The program structure can be divided into two different parts: the start up and the running mode. During the start up the system run trough different initializations:

1. Initialize the system clock
2. Initialize servo outputs
3. Initialize SPI communication with MCU B



Available Link Objects:
libc.a
libm.a
libobjc.a
libprintf_flt.a
libprintf_min.a
libscanf_flt.a
libscanf_min.a

Link with These Objects:
libm.a

Figure 4.6: Project Options

4. Initialize debug USART(optional)
5. Initialize IMU and wait for data
6. Initialize SIMULINK controller
7. Initialize GPS and wait for approved GPS quality
8. Initialize the NED frame.

Most of these initialization routines are presented in Chapter 3. However, the GPS quality check and the NED frame initialization is not. When the GPS initialization is done, the system waits for valid GPS data to be made available so the NED frame origin can be determined. The system is currently set up to use the starting position of the vehicle as the centre of origin. One problem is that if the first GPS measurement is used, there is no guarantee that is an accurate one. Most likely it is a rather inaccurate one derived from signals provided by a minimum of GPS satellites. To increase the probability of a good position measurement a lower threshold is set on the number of satellite signals used to calculate the position. This threshold is currently set to seven, which is about half the amount of satellite signals possible to track on the GPS receiver. In addition ten measurements are added together and used to find an average position which is used as the origin. This last precaution weakens the influence of one inaccurate measurement giving an even higher probability for a good position measurement used as the origin.

Figure 4.7 is flowchart showing the program flow after the initializations have been completed. The controller block represents measurement gathering, controller computation and application. It also includes the ellipsoidal ECEF coordinates to NED conversion. An iteration of the controller is executed in the following way

```
controller_U.gps[0]=delta_N;
controller_U.gps[1]=delta_E;
controller_U.wp[0]=wp[i][0];
controller_U.wp[1]=wp[i][1];
controller_U.wp[2]=wp[i][2];
controller_U.wp[3]=wp[i-1][0];
controller_U.wp[4]=wp[i-1][1];
controller_U.wp[5]=wp[i-1][2];

double yaw=data->euler_yaw;
controller_U.imu[2]=yaw;;

MdlOutputs();

SERVO_1_SETPOS((char)controller_Y.Steering);
```

Since the IMU measurement are interrupt driven, putting this data into the controller input structure is the last operation done before computing the controller outputs. This is to use the newest data possible in the controller algorithm.

The *MdlUpdate* function is not included since it in the current version does not contribute to anything. It must be included if integral effect are added to any of the controllers. A discrete integration block has been tested on the XMEGA 265A3 with successful results. The discrete integrator is updated in the *MdlUpdate* function. Since it has to be called each time step it should be included in a Timer/-Counter interrupt routine to guarantee that the integrated values are correct.

The *waypoint check* block in Figure 4.7 represents check of the *next* variable returned by the controller. If it is 1, the waypoint pointer is incremented and the next waypoint is sent to the controller.

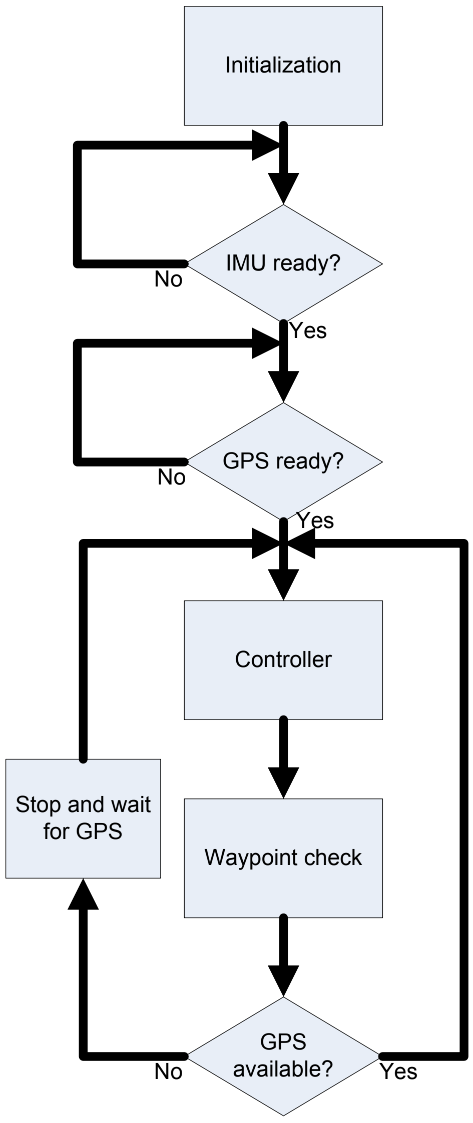


Figure 4.7: Local Bug Program Flow

5 System Test and Test Results

This chapter will present the set up and results of the tests conducted on the Local Bug Platform. The data logs where imported to Matlab and plotted in the same plot as the desired trajectory. Appendix A.3 presents the code used to import and plot the test results.

5.1 Test Area

The tests needed to be conducted on a location with reasonable good GPS coverage. It also should be flat and protected against unforeseen disturbances. A tarmac covered square located near Kristiansten fortress in Trondheim satisfied the desired conditions. The square is about 40×40 metres and is surrounded by relatively flat lawn in approximately the same level as the tarmac. If the vehicle is not able to stay on the dedicated test area it will still be able to continue and complete the route without any critical change in the driving conditions. It is also isolated from tall buildings and has a large amount of open areas in most directions.

5.2 Test Route

The waypoints where chosen to create a route challenging the control system with sharp turns. In addition the route is designed to keep the vehicle within the tarmac covered area. The vehicle was placed five metres from both tarmac edges in the south east corner during the initialization in order to define this as the origin of

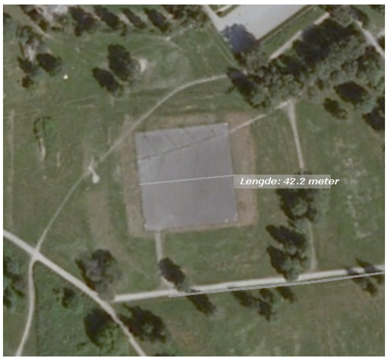


Figure 5.1: Test Area

the NED frame. Five waypoints defines the route, however the first and the last waypoint is at the origin. The first waypoint is not used as a standard waypoint;

Nr.	x_n	y_n	r_a
0	0	0	2
1	30	0	2
2	0	-30	2
3	30	-10	2
4	0	0	2

Table 5.1: Waypoints used in the Local Bug test. r_a denotes the radius of acceptance

it is only used in the Cross-Track Error algorithm to define the first line to follow. Hence it is listed as number zero. r_a was set to two metres, giving the controller quite good margin on the waypoint accuracy. The sharp turns are used to challenge the control system. When a turn is sharper than the vehicles maximum turn rate, it provokes the differences between the two algorithms used and requires the controller to utilize vehicles turning capabilities.

5.3 Simulation

The Local Bug simulator has been used to simulate the test trajectory, making it possible to compare the kinematic model to actual behaviour. Figure 5.2 is the plotted results from the simulation conducted with the LOS guidance algorithm. When the vehicle reaches the edge of the circle of acceptance, it immediately changes heading and heads directly for the next waypoint. Because the circle of acceptance has a radius of two metres the vehicle cuts the turn short. The first turn is exactly as sharp as the maximum of what the vehicle can handle. This causes the vehicle to follow the straight line to the next waypoint.

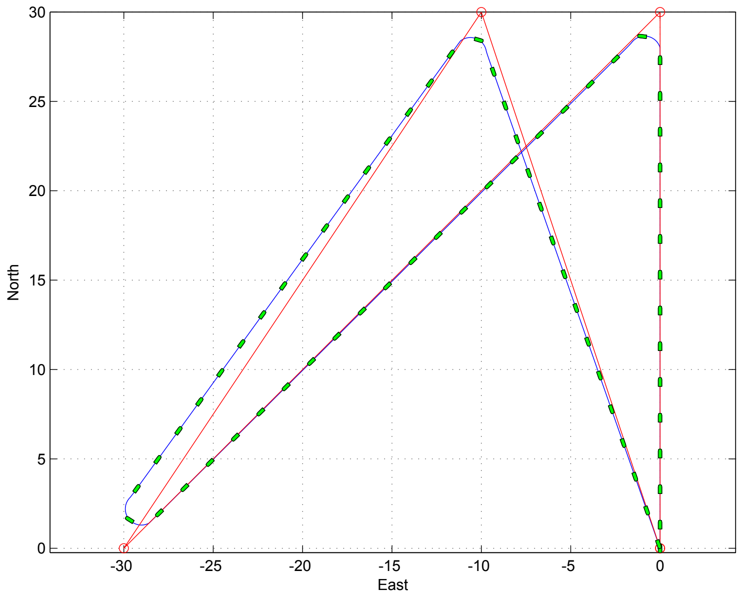


Figure 5.2: Simulation of the test with the LOS algorithm plotted in the NED frame. North(m) and East(m) are x_n and y_n respectively.

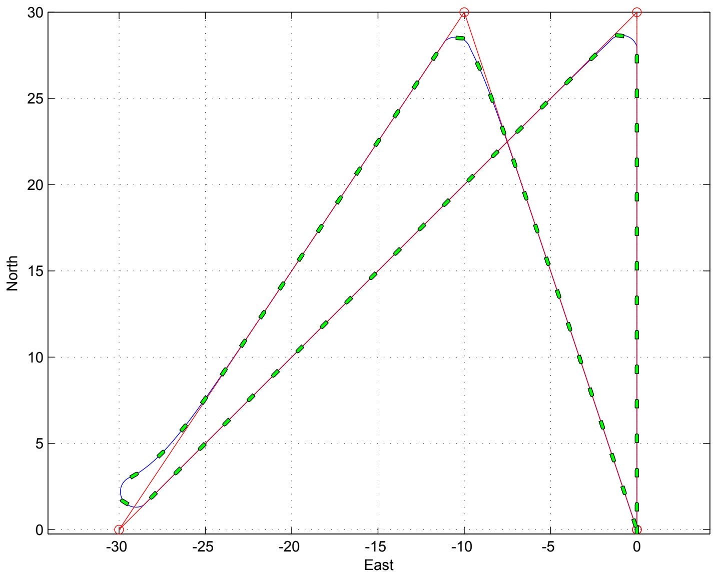


Figure 5.3: Simulation of the test with the Cross-Track Error algorithm plotted in the NED frame. North(m) and East(m) are x_n and y_n respectively.

At the second waypoint change, the turn is sharp enough to give the vehicle an overshoot. This happens even though the turn has already been cut causing the required turn radius to be larger than it would be otherwise. Notice that the turn rate required by a turn depends on the radius of acceptance. Larger radius gives a lower required turn rate and vice versa.

Figure 5.3 is the results from a simulation done with the Cross-Track Error algorithm. In comparison to the LOS simulation, the CTE always converges to desired path, minimizing the Cross-Track Error. At the second waypoint change it quite evident that the overshoot is compensated for and the vehicle behaves as desired.

5.4 Line of Sight Test

The vehicle was placed at the position of the desired NED origin and powered up. When the system was ready to go the autopilot were switched on. The very first test log is plotted in Figure 5.4. All waypoints were visited and the test was successfully terminated at the last waypoint. However, the vehicle does not head straight for the next waypoint. A more arced path is taken indicating either inaccurate measurements or bad controller performance. Considering the heading of the vehicle in relation to the path taken by the vehicle, it is evident that either the GPS or the IMU measurements are inaccurate.

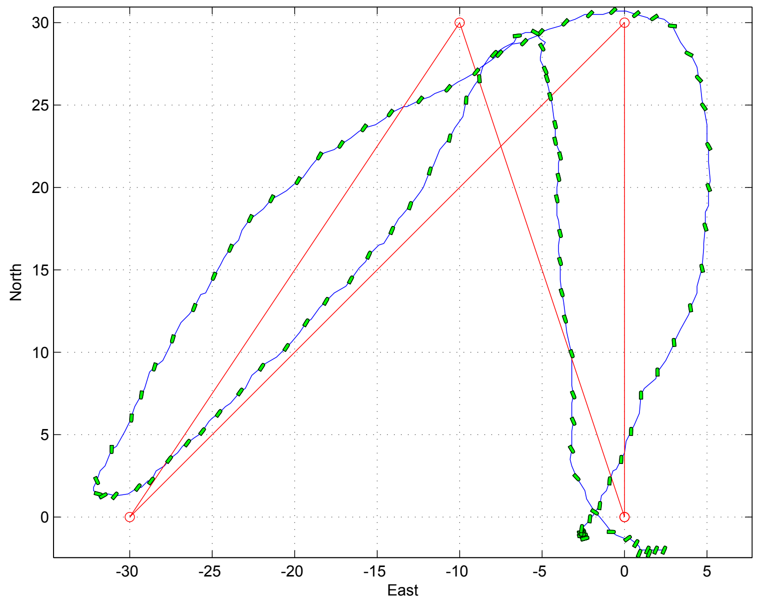


Figure 5.4: The first test with the LOS algorithm plotted in the NED frame. North(m) and East(m) are x_n and y_n respectively.

On the way to the first waypoint the vehicle's heading is pointing at or close by the next waypoint. However, the vehicle is measured to be moving partially sideways indicating that the position measurements do not correspond to the heading. If Figure 5.4 is seen in relation to the video file *testLOS1* located on the CD in Appendix C the behaviour do correspond quite well. The distance between start and the first waypoint is clearly covered by travelling an arced path. Hence the controller is not as aggressive as desired. The same test was conducted multiple times yielding more or less the same results. Most times the sideways sliding sensation was not present. However the paths taken between waypoints still tended to be arced.

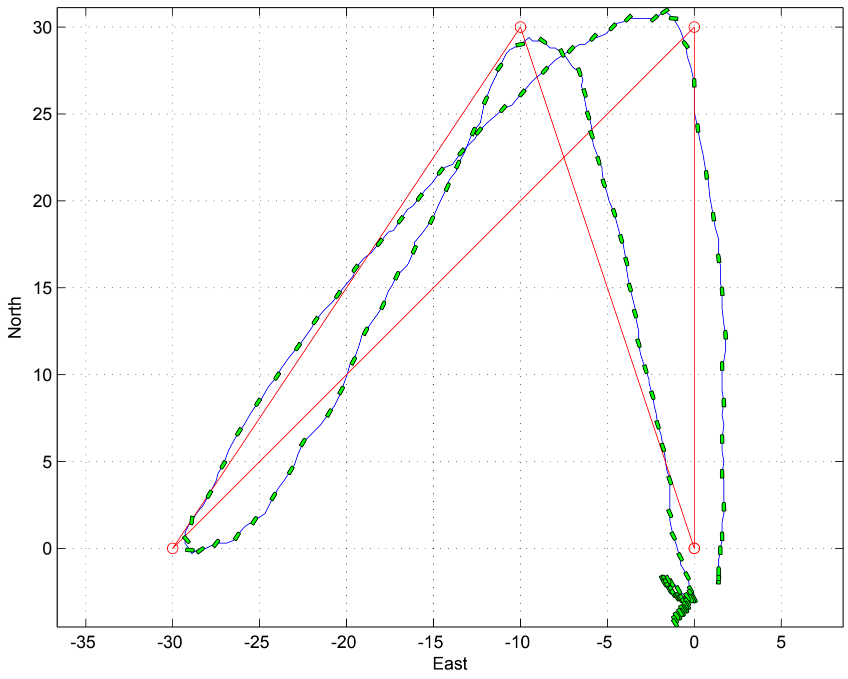


Figure 5.5: The second test with the LOS algorithm plotted in the NED frame. North(m) and East(m) are x_n and y_n respectively.

The heading controller gain was increased in order to make it more aggressive. Figure 5.5 is the test log with the controller gain set to three. The vehicle is now much closer to taking a heading straight for the next waypoint. When the vehicle overshoots a turn it recovers and aims almost directly at the next waypoint. Multiple test runs were made to confirm the improved behaviour. However, only one is included her to illustrate the difference in performance between the two test setups.

5.5 Cross-Track Error Test

The heading controller gain used in the CTE tests was the same as in the last LOS test, since this clearly improved the performance. Figure 5.6 is a plot of the log data from this test. The vehicle completed the test route with satisfactory performance. It is not spot on the desired path during the test, but this cannot be expected either. At the first waypoint change the vehicle overshoots the desired path with approximately five metres before it converges to the straight line and heads for the next waypoint. *testCTE1.avi* in Appendix C is a video recorded of the test and it shows the same behaviour as plotted in the figure.

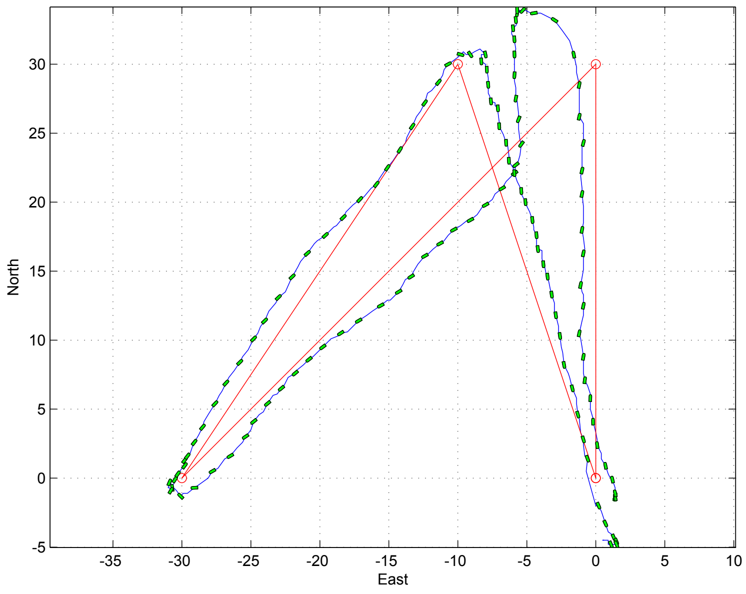


Figure 5.6: The first test with the CTE algorithm plotted in the NED frame. North(m) and East(m) are x_n and y_n respectively.

To test the systems ability to converge to the desired path, it was moved after the initialization of the NED frame. The vehicle was now staring at approximately four metres east and ten metres north with a heading angle $\psi = 230°$. When commencing the test, the vehicle is headed for the origin and must turn around and follow the desired path north to the next waypoint. Figure 5.7 is the plot of this test. It shows the vehicle turning around and heading north to the first waypoint. The path following in this test was much more satisfying than in the previous ones. It only deviates one metre from the desired path on the straight lines and quickly recovers from overshoots at the waypoint changes.

Log files and videos from these tests are included on the CD in Appendix C.

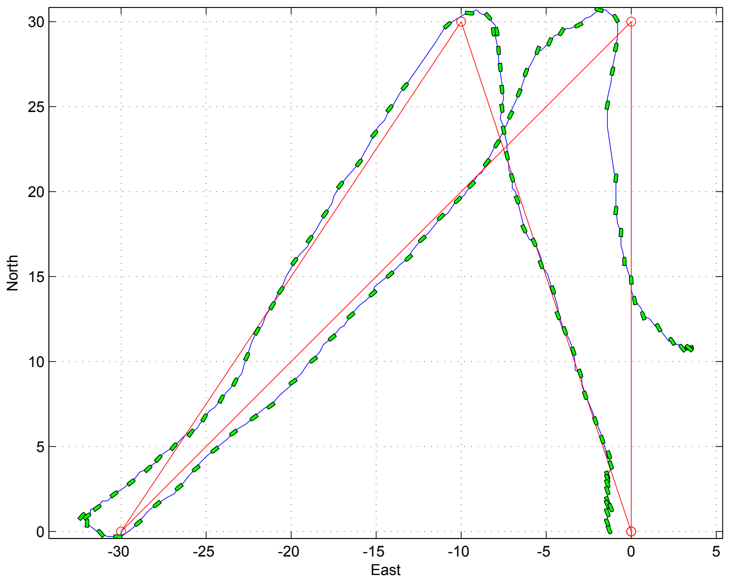


Figure 5.7: The second test with the CTE algorithm plotted in the NED frame. North(m) and East(m) are x_n and y_n respectively.

6 Discussion

Having presented and discussed different models in Chapter 2. Used this theory to implement a simulator able to test the functionality of the control system presented in Chapter 4. Combined the controller with the framework presented in Chapter 3 and tested the system in Chapter 5. Some comments about the overall system and individual system parts are presented in this chapter.

Model and Simulator

The vehicle models presented in Chapter 2 are used in the Local Bug simulator. Because of inaccurate tyre models the rigid body model does not work properly and can currently not be used to simulate the vehicle behaviour. The kinematic model can be used to conduct simulations and yields a quite accurate simulation under conditions where the model assumptions are satisfied. Measurements fed to the controller block are created by transforming the model outputs into the frame used by the measurement units. This gives controller measurements that correspond to the real measurements, but still are somewhat different. The simulated measurements are 100 percent accurate, while the real measurement's accuracy may vary. Inaccuracy can be added to the measurements by adding a noise and some amount of random deviations. Such changes can be done in the future if needed. During this thesis the simulator has been used to test if the controller worked, and managed to get the vehicle trough a test route. Once the controller is found to work more extensive and accurate testing can be done on the Local Bug platform. This is after all the main reason for developing such a platform.

Hardware

The Phoenix II can be mounted with either side up, to provide the possibility of choosing which side can be easily accessed. It would however been convenient having is mounted vertically making it possible to access both sides without having to loosen the Phoenix II from the radio box. To do so the radio box would have to be rebuilt to create more vertical space. This would be the most convenient solution when using the vehicle outdoor. Alternatively new versions of the Phoenix could be designed to have certain features located on one side e.g. the log switch, status LEDs, SD card slot and the programming interfaces for MCU A and B. There are of course limitations to the electronic PCB design that might make it impossible or cumbersome to achieve this layout. It should however be added to the list of desired features for the next version.

The Radiocrafts antenna connector is currently used to mount the Phoenix II to the radio box. This makes the connector vulnerable to vibrations caused by rough driving conditions. It would be desirable to hold the Phoenix in place by several mounting points holding the PCB in place. However, since the Phoenix II is designed to be as small as possible, there are close to no space for attaching clips or other mounting devices, not to mention drilling holes and use screws and nuts to mount it. Fortunately the antenna connector is thoroughly soldered to the PCB card and seems to be able to tolerate enough vibrations for this to be a appropriate mounting solution for the Local Bug.

Protecting the IMU from the magnetic fields generated by the electric motor was the primary objective when the IMU was placed on the Local Bug. The magnetometer can be affected by magnetic fields causing the IMU's north to deviate from true north. This limited the position alternatives to the front segment of the vehicle. In order to protect the unit from external dangers it was mounted behind the front axle on top of the radio box containing the GPS receiver and the Phoenix II. The heading measurements were not found to be affected by the motor, not even at full speed.

Software Framework

The framework designed for Phoenix II handles communication with measurement units used on the Local Bug. Measurements are transformed into the frames and dimensions required by the controller. In order to utilize as much as possible of the Phoenix II design and keep the Local Bug compatible with the overall Local Hawk project, most of the drivers are based on the beta drivers for the Phoenix II.

The IMU communication is stable and the measurement update is controlled by a timer interrupt insuring that the latest measurements are made available to the controller algorithm. To ensure correct configuration of the IMU, it is reconfigured on every reboot of the Phoenix II. Some problems have been encountered when the IMU parser was running only when measurements were needed. The IMU returned the same value no matter how it was oriented. When the parser was controlled by a timer interrupt the communication was stable throughout this process.

Currently, the *mdlUpdate* function is left unused. This is because it does not contribute to anything since there are no states in need of updates on a timely basis. If a integrator is implemented in the controller, it would be necessary to call *mdlUpdate* each time step. The time step is set in the SIMULINK diagram before generating the C-code. A timer interrupt is a suitable way to time the frequency it is called at. It can be implemented as done with the IMU parse function in Section 3.2.5.

The Guidance Principals

Both LOS and CTE guidance successfully fulfilled the principal goals of a guidance system, reaching all waypoints. However there are weaknesses to both approaches. The LOS algorithm may miss the circle of acceptance of a waypoint if it has a large cross-track error relative the imaginary line between the last waypoint and the next. This will make the vehicle circulate the waypoint until the acceptance criteria is fulfilled.

The CTE algorithm implemented uses the along path distance variable to determine when change waypoints. This is a less demanding condition as it will allow waypoint changes without actually evaluating the proximity to the waypoint. The vehicle will not circulate around one waypoint as it is allowed to switch to the next as soon as it has passed the along path distance defined for the waypoint. This is no problem as long as the controller manages to follow the straight line. Considering the Local Bug these problems does not cause any concern as the disturbances are limited and hence the controller task is less demanding. However, this thesis constitutes a part of the Local Hawk project where the overall goal is to design an AUAV. When considering aerial vehicles the disturbances are much harsher and challenge the controller more than on land-based vehicles. By using the along path distance switch condition assumes that the controller are able to follow the path given.

Implementation and Performance

The Controller used on the Local Bug is designed in SIMULINK and used to generate C-code by use of Real-Time Workshop. This approach was chosen in order to ease the work of designing a controller and to gain correlation between controllers tested in the simulator and the ones implemented on the test platform. An alternative approach would be to write the controller C-code manually based on the one implemented in SIMULINK. This would give the control designer more freedom in how to access controller functions and utilize the computational power supplied by the Phoenix II. However, there would be no guarantee that the controller functionality tested in SIMULINK would be exactly the same as the controller implemented on the vehicle.

The controller designed in SIMULINK is currently only controlling the vehicle heading. Considering the test results presented in Chapter 5 it is evident that the controller performance is dependent of the measurement stability. This is also quite intuitive; if the vehicle position cannot be determined it is impossible to find a heading pointing to the next waypoint. Less accurate positions yield less accurate desired headings. Since the log data is the GPS data this is not necessarily the current position of the vehicle. As mentioned in Section 3.2.4, the GPS unit only guaranties that 50 percent of the measurements are within 2.5 metres of the actual position.

In Figure 5.6 the vehicle has a stationary position deviation. It is consequently positioned left of the desired path. Adding integral effect to the heading controller, making it a PI-controller, can contribute to remove this stationary deviation(Balchen et al., 2003).

7 Conclusion

In this thesis two different car models are presented and evaluated. One of them is implemented as the vehicle model of the Local Bug simulator making it possible to test vehicle control and guidance principals without actually implementing them on a vehicle. The model chosen to be implemented is a kinematic bicycle model which is derived purely from geometrical aspect. SIMULINK are used to realize the simulator, by running the *run.m* file the model is initialized and simulated. The resulting behaviour is represented in the NED frame and displayed in a 2D plot.

A R/C car is used as the base when creating a test platform for the Local Hawk project. This test platform is referred to as the Local Bug. It consists of a Savage XL R/C car, with a GPS receiver and an IMU Unit used to measure its position and orientation. Phoenix II autopilot platform were designed and produced in a parallel Local Hawk project, this unit is used to tie the measurements units and the vehicle together. All components are mounted in the vehicle at suitable places when considering protection from both electromagnetic fields and physical dangers such as collisions.

Drivers and basic software functionality is implemented in C-code to form a software framework running on the Phoenix II. This framework handles communication with external measurement units and controls the vehicles manipulated variables, throttle and steering angle. It is set up to interact with Real-Time Workshop generated controller functions used to calculate the desired steering angle from waypoint data.

A guide is provided on how to use Real-Time Workshop to generate generic C-code from a controller implemented in SIMULINK and how to relocate, include and compile this code a part of the total Local Bug software package. This approach where found to be a very effective way of realizing the controllers tested in SIMULINK. It is also found to be sufficiently effective, performance wise, to be used to design controllers for the system.

A controller and two guidance systems are presented, implemented and successfully tested on the platform. Both guidance algorithms fulfilled the primal goal of completing a test path defined by waypoints. The overall system has a satisfactory performance and the Local Bug can now be used in the Local Hawk project to test alternative controllers and guidance principals.

7.1 Future Work

The simulator implemented in SIMULINK is based on a kinematic model. It is assumed that the wheels only moves in the direction of the tyre track, in other words with no side slip. If the vehicle is to be used on slippery driving conditions this assumption does not hold. A model based on force computation should be used as the simulator needs to challenge the controller in a similar way as slippery roads will. The main challenge in using such a model is the tyre model calculating the forces acting in the lateral wheel direction. Such models can get really complex and include everything from rubber surface friction to tyre deformation, but are necessary to some extension to simulate the vehicles behaviour under various driving conditions. For higher speed, more degrees of freedom needs to be taken into account. Hard turns during high speed can make the vehicle flip over. This takes the model to a whole new level, 3D modelling with 5DOF or 6DOF gets quite a bit more complicated.

Under such circumstances the controller should also be improved taking more states into account when computing the control variables. It should for instance measure roll angle and use it to limit the steering angle in order to keep the vehicle from flipping over. Alternatively a overall more advanced controller could be implemented. The limited amount of computational power on the Phoenix will probably make advanced adaptive and non-linear controllers impossible to use, but some of the functionality might be added. Before taking the controller to these extremes it is more important make the best of the basic controllers. Integral effect should be added to the controller to decrease the size of the stationary deviations. As mentioned in Chapter 4, this can be done by including the Real-time Workshop function *mdlUpdate* to a timer interrupt and design a PI-controller realizing the integral term with a discrete integral block in SIMULINK. Some of the Phoenix II drivers and parsers should be improved making more measurements available to the controller. Velocity measurements would make it possible to implement a speed controller and perhaps include break before turning functionality to the guidance block, and thereby reduce the risk of flipping over.

The Phoenix II is equipped with a Radiocrafts chip allowing it to send and receive data trough a Radio Frequency (RF) link. This unit is not used in the current version of the Local Bug. It can add a great deal of functionality to the platform e.g. real-time waypoint updates and data logging and should be put to use in later versions of the Local Bug.

Appendices

A
Drivers and Routines

A.1 IMU: Initialization Routine

```
void IMU_init(void){
uint8_t data[ 256 ];
PORTF.DIR|=0x01;
 IMU_Lamp_Off();
int init=1;
printf("\n\rIMU init");
XUSART_INIT3( XUSART_IMU );
MTComm_Initialize( &imu, &XUSART_IMU, &XUSART_IMU.rxBuffer );

while(init){
 if((tmp = MTComm_Parse( &imu ))!= 0 ){;
  if ( tmp == MTCOMM_MID_WAKEUP ){
     MTComm_SendMessage( &imu, MTCOMM_MID_WAKEUPACK, 0, 0 );
      _delay_ms( 250 );
    MTComm_SendMessage( &imu, MTCOMM_MID_GOTOCONFIG, 0, 0 );
      _delay_ms( 500 );

    /* Set output mode */
    uint16_t * outputMode = data;
    *outputMode = MTCOMM_OUTPUTMODE_CALIBRATED_bm
                           | MTCOMM_OUTPUTMODE_ORIENTATION_bm;

      MTComm_SendMessage( &imu, MTCOMM_MID_SETOUTPUTMODE, data, 2 );

       _delay_ms( 250 );

      /* Set output settings */
      uint32_t * outputSettings = data;
      *outputSettings =
        MTCOMM_OUTPUTSETTINGS_TIMESTAMP_SAMPLECOUNTER_gc
       | MTCOMM_OUTPUTSETTINGS_ORIENTATION_EULER_gc
       | MTCOMM_OUTPUTSETTINGS_ACCELERATION_DISABLE_bm
       | MTCOMM_OUTPUTSETTINGS_MAGNETOMETER_DISABLE_bm
       | MTCOMM_OUTPUTSETTINGS_OUTPUTFORMAT_FIXED1220_gc
       | MTCOMM_OUTPUTSETTINGS_COORDINATES_NED_bm;
      MTComm_SendMessage( &imu, MTCOMM_MID_SETOUTPUTSETTINGS, data, 4 );

       _delay_ms( 250 );

       /* Set output period */
       uint16_t * period = data;
      *period = MTCOMM_PERIOD_100HZ;
       MTComm_SendMessage( &imu, MTCOMM_MID_SETPERIOD, data, 2 );

       _delay_ms( 250 );
       MTComm_SendMessage( &imu, MTCOMM_MID_GOTOMEASUREMENT, 0, 0 );
         _delay_ms( 500 );

      }else if ( tmp == MTCOMM_MID_CONFIGURATION ){
            _delay_ms( 250 );
            MTComm_SendMessage( &imu, MTCOMM_MID_RESET, 0, 0 );
   }else if ( tmp == MTCOMM_MID_MTDATA )
   {
            IMU_Lamp_On();
            init=0;

   }

  }
 }
}
```

A.2 MCU B: Writing Log to Micro SD Card

```
/* Checks if a full struct is plased in the rxbuffer + MID + lenght */

if ( XBuffer_Count( &xspi.rxBuffer ) == sizeof( log_message ) + 2 )
{
/*If the mid and lenght is correct the message is readdy to be read out*/
  XBuffer_RemoveHead( &tmp, &xspi.rxBuffer );
  if ( tmp == sizeof( log_message ) + 1)
    {
  XBuffer_RemoveHead( &tmp, &xspi.rxBuffer );
  if ( tmp == MID_LOG_MESSAGE )
  {
  p = ( uint8_t * ) &incomingMessage;
  for ( i = 0; i < sizeof( log_message ); i++ )
  {
   XBuffer_RemoveHead( p++, &xspi.rxBuffer );
  }
/*  If the log switch is active and the logging is currently inactive
    the SD communication is set up and logging is initiated.*/
    if(PORTF.IN&0x08)
  {
   Log_Lamp_Off();
   if(isLogging)
   {    printf("stopps\n\r");
    isLogging=false;
    f_close(&fil_obj);
    f_mount( 0, 0);
   }
   }else
   {
        Log_Lamp_On();
    if(!isLogging)
    {
     printf("logging\n\r");
     isLogging=true;
     f_mount(0, &myfat);
     f_open(&fil_obj, "test.txt", FA_WRITE | FA_OPEN_ALWAYS);
     f_printf(&fil_obj, " %d %d %d \n",(int)incomingMessage.north,
       (int)incomingMessage.east, (int)incomingMessage.yaw );
    }else
    {
     f_printf(&fil_obj, " %d %d %d \n",(int)incomingMessage.north,
       (int)incomingMessage.east, (int)incomingMessage.yaw );
    }
   }
  }
 }
}
```

A.3 Matlab Script for Plotting Local Bug Data Log

```
close all
clear all
%define waypoints used
wp=[[0,0,1],[30,0,1],[0,−30,1],[30,−10,1],[0,0,1]];

%load log data from file
log=load('los3t.txt');

%plot waypoints with straigth lines between them
plot(wp(2:3:length(wp)),wp(1:3:length(wp)), '−or');
hold on;

%plot the log data
plot(log(100:length(log)−200,2)/10, log(100:length(log)−200,1)/10);

%set how frequently the vehicle should be plotted. 1, print on every
%log data available, 2, on every second and so on.
    step=3;

%create list of indexes when the vehicle is going to be plotted
    index=[];
  index=1:step:length(log);
%length of vehicle
L=0.5;
%plot 2D figure for the indexes in the list
for now=index
     tmpR=[cos(2*pi−log(now,3)*pi/180) −sin(2*pi−log(now,3)*pi/180);
            sin(2*pi−log(now,3)*pi/180) cos(2*pi−log(now,3)*pi/180)];

     car = tmpR*[L/2 .9*L/2 .5*L/2 −L/2 −L/2 .5*L/2 .9*L/2 L/2;
              0 0.08 0.1 0.1 −0.1 −0.1 −0.08 0];

     plot(log(now,2)/10+car(2,:),log(now,1)/10+car(1,:),'g');
     patch(log(now,2)/10+car(2,:),log(now,1)/10+car(1,:),'g');
end
%name axis
grid on
xlabel('East')
ylabel('North')

axis('equal');
```

B
The Local Bug Assembly Pictures

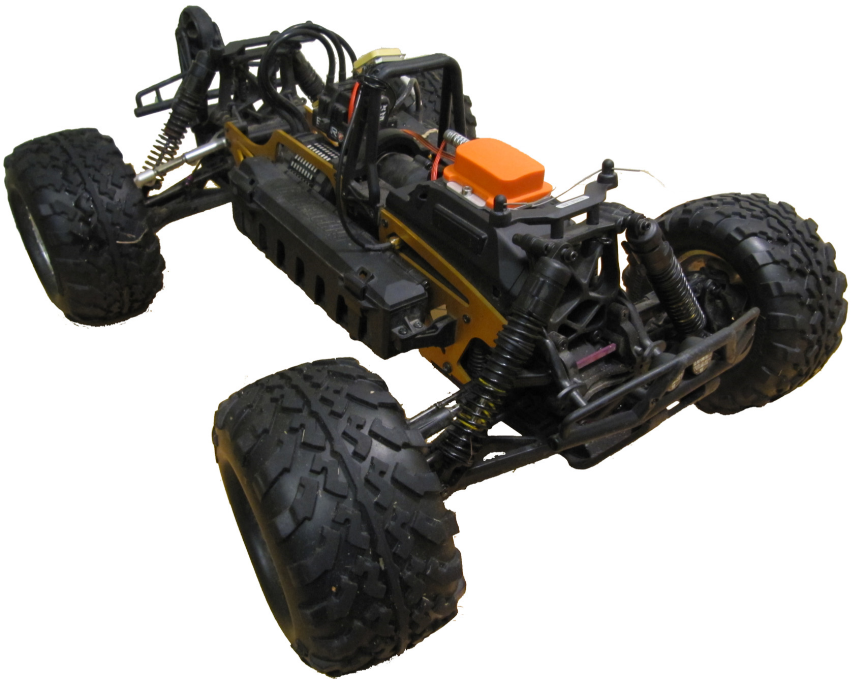


Figure B.1: The Local Bug fully assembled

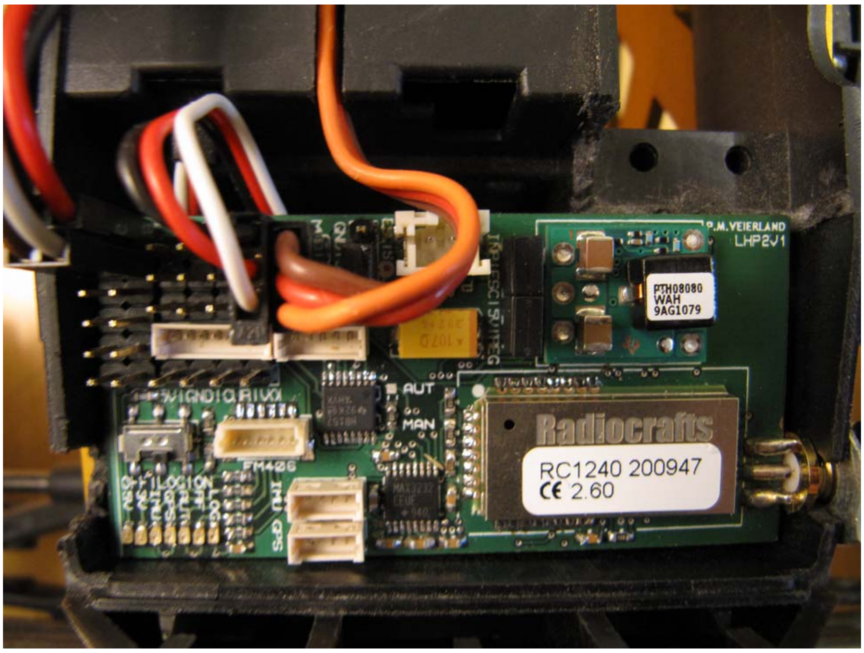


Figure B.2: The Phoenix II mounted in the radio box

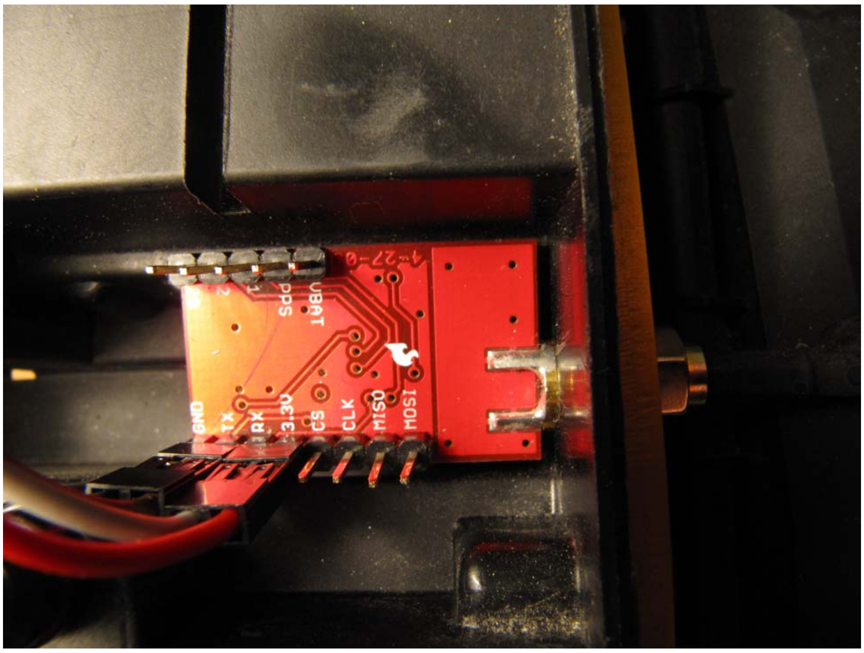


Figure B.3: The GPS Unit is mounted in the rear compartment of the radio box.

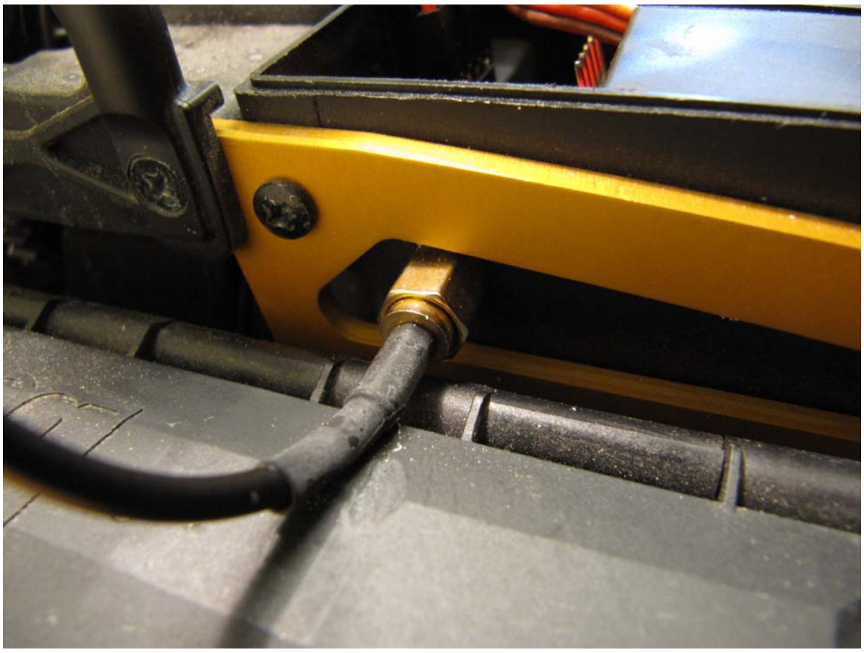


Figure B.4: The GPS antenna is attached to the GPS unit trough a hole in the chassis holding the GPS unit in place.

Bibliography

Ackermann, J. (1993). *Robust Control.* Springer-Verlag.

ATMEL (2010). XMEGA A3 Microcontroller. `http://www.atmel.com`.

Balchen, J. G., T. Andresen, and B. A. Foss (2003). *Reguleringsteknikk.* Department of Engineering Cybernetics, Norwegian University of Science and Technology.

Chris H (2010). `http://www.basementcode.com/avr/sd_fatfs/fatfs.php` FatFs explained for AVRs and SD cards.

Clynch, J. R. (2006). Radius of the Earth - Radii Used in Geodesy. Department of Oceanography, Naval Postgraduate School.

Drake, S. P. (2002). Converting GPS Coordinates ($\phi\lambda h$) to Navigation Coordinates (ENU). DSTO Electronics and Surveillance Research Laboratory.

Edwards, C. H. and D. E. Penny (2002). *Calculus.* Prentice Hall, Pearson Education Inernational.

Egeland, O. and J. T. Gravdahl (2002). *Modeling and Simulation for Automatic Control.* Marine Cybernetics.

Elm Chan (2010). `http://elm-chan.org/fsw/ff/00index_e.html`.

Forssell, B. (2003). *Radionavigation Systems.* Tapir Akademiske Forlag.

Fossen, T. I. (2002). *Marine Control Systems.* Marine Cybernetics.

GPS Viewer (2010). `http://www.sparkfun.com/datasheets/GPS/Modules/GPS%20Viewer_0506.zip`.

Hagen, S. W. et al. (2009). Local Hawk Summer Project. Technical report, Kongsberg Gruppen.

HPi Racing Webpage (2010). `http://www.hpiracing.com/`.

Miljeteig, L. I. et al. (2008). Local Hawk, Summer Project. Technical report, Kongsberg Gruppen.

Pacejka, H. B. (2006). *Tire and Vehicle Dynamics.* SAE International.

Petersen, I. (2003). *Wheel Slip Control in ABS Brakes using Gain Scheduled Optimal Control with Constraints.* Department of Engineering Cybernetics, Norwegian University of Science and Technology.

Rajamani, R. (2006). *Vehicle Dynamics and Control.* Springer.

Reza, J. N. (2009). *Vehicle Dynamics: Theory and Applications.* Springer.

Rottmann, K. (2003). *Matematisk Formelsamling.* Spektrum forlag.

SD Group (2006). Sd specifications: Physical layer simplified specification. `http://www.sdcard.org/developers/tech/sdcard/pls/Simplified_Physical_Layer_Spec.pdf`.

Setiawan, J. D., M. Safarudin, and A. Singh (2009). *Modeling, Simulation and Validation of 14 DOF Full Vehicle Model.* Faculty of Mechanical Engineering, University Teknikal Malaysia Melaka, Malaysia.

Skandan, N. (2005a). Rtw c code how to use matlab c code in your application. `http://www.mathworks.com/matlabcentral/`.

Skandan, N. (2005b). Step by step approach to exporting rtw code generated by generic real time target to your application. `http://www.mathworks.com/matlabcentral/`.

SKYTRAQ (2009). The venus634flpx 65 channel low power gps receiver data sheet.

The MathWorks, I. (2010). Real-time workshop 7, user's guide.

Tipler, P. A. and G. Mosca (2004). *Physics for Scientists and Engineers.* W. H. Freeman and Company.

Tjønnås, J. and T. A. Johansen (2010). Stabilization of automotive vehicles using active steering and adaptive brake control allocation. *IEEE Trans. Control Systems Technology*.

Veierland, P. M. (2010). Local hawk phoenix ii. Technical report, Department of Computer Science, Aberystwyth University.

Vik, B. (2009). *Intefrated Satellite and Inertial Navigation Systems.* Department of Engineering Cybernetics, Norwegian University of Science and Technology.

Vold, J. Ø. (2009). Guidance System of an AUAV - Local Hawk. Technical report, Department of Engineering Cybernetics, Norwegian University of Science and Technology.

Xsens (2006). MTi and MTx User Manual and Technical Documentation.

Xsens (2008). MT Low-Level Communication Protocol Documentation.

Zhao, Y. (1997). *Vehicle Location and Navigation Systems.* Artech House, Inc.