Doppler Radar Speed Measurement Based On A 24 GHz Radar Sensor

Joacim Dybedal

Master of Science in Electronics
Submission date: June 2013
Supervisor: Morten Olavsbråten, IET
Co-supervisor: Anders Hagen, Q-Free

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Abstract

This thesis will present the implementation of an on-board speed measurement system using a single 24.1 GHz Doppler radar sensor and specialized algorithms to measure the true speed of a vehicle. Two different algorithms are implemented, the first based on estimating the Doppler power density spectrum and extracting the strongest frequency component, and the second based on correlation between the Doppler spectrum and pre-estimated theoretical spectra. The output can be displayed to the user in real-time as well as stored for future reference.

An ARM Cortex M4 microcontroller with digital signal processing capabilities is used as the hardware platform, with an audio CODEC chip used as the analog to digital converter. The software is implemented using the C programming language.

The system is tested and the measurements are compared to a GPS reference system, with results showing statistical mean errors down to as little as 0.03 km/h and -0.18 %, and a standard deviation of 0.87 km/h during the final test runs.

Preface

This thesis was written as part of completing the Master's degree in Electronics at the Department of Electronics and Telecommunications (IET) at the Norwegian university of science and technology (NTNU).

During the six months in which this thesis was written I have gained much new knowledge, especially in the fields of radar technology and signal processing, but also about the challenges one is faced with when implementing a large project using unfamiliar tools and equipment. My time at NTNU have been one of the best periods in my life, and I leave with experience and knowledge that will be invaluable in the forthcoming years.

I would like to express my gratitude to my supervisor Morten Olavsbråten at IET and to my co-supervisor Anders Hagen at Q-Free ASA for all the help and support they have given me during these six months. A special thanks also goes out to the rest of the staff at Q-Free ASA that were involved during the testing of the system.

Finally, I would like to thank the fellow students at the study room, Magne, Lars and Jonathan, for making this last semester at NTNU both interesting and fun.

Trondheim, June 12, 2013

Joacim Dybedal

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

1.1 Motivation

The problem solved in this thesis was presented by Q-Free ASA, a Trondheim company focusing on solutions for Road User Charging and Advanced Transportation Management. The problem is based on a need for a system for speed measurements that is independent of other systems such as the speedometer and GPS, that will be used in systems where the traveled distance is used as a base for charging rates. Hence, a solution based on a Doppler radar system was proposed.

1.2 Problem Description

This section will give a description of the problem which is to be solved in this project.

1.2.1 The Problem Text

The given problem text reads as follows:

"Doppler radars are used for many applications, like door openers, range measurement, speed radars etc. The radar microwave unit consist of a homodyne transceiver, one path of the local oscillator is transmitted through an antenna the other path is feed to a mixer, the RF input of the mixer is connected to an other antenna, receiving the reflected signal. The IF output (the mixer product or quadrature I and Q output) is the input to the signal processor. The RF frequency is 24GHz. The task is to design a light embedded signalprocessing SW to make reliable speed measurements. The HW is available, based on an available RF unit and a microcontroller design-kit."

1.2.2 System Specifications

As the problem text does not specify any formal specifications that the speed measurement system has to comply with, some specifications were developed in the initial stages of the project. These specifications were made to set some boundaries for the project, as well as to set a goal for what the project hopes to accomplish.

- The system should be able to measure speeds from 0 to 250 km/h.

- The system should be able to measure speeds with a resolution of at least 0.5 km/h, and with the best possible accuracy.
- The system should be able to update the speed measurements at intervals no longer than 1 second.
- The system should be able to present the measurements in real-time, as well as logging the speed data for later use.

1.3 Speed Measurement Systems

This section will give short presentations of some of the various solutions for speed measurements that already exists, and some previous approaches using radar.

1.3.1 The Speedometer

The speedometer is by far the most widespread type of speed measurement system, as it exists in nearly all motorized vehicles that are allowed to use public roads. There are different types of speedometers, and as an example, an electronic speedometer works by interpreting the pulse signal generated from a rotating magnet mounted on the drive shaft connecting the transmission to the tires. The frequency of this signal represents the speed of the vehicle [4].

Because of how the measurement is performed, the indicated speed will vary with the size of the tires used on the vehicle, air pressure of the tires, etc.

Accuracy Speedometers mounted on modern cars are subject to regulations from different authorities, e.g. national governments, the United Nations Economic Commission for Europe and the European Union. The European Council Directive 75/443/EEC [5] states that the following relationship between the true speed (V_2) and the indicated speed (V_1) must be satisfied:

$$0 \leq V_1 - V_2 \leq \frac{V_2}{10} + 4\text{km/h} \tag{1.1}$$

Hence, the indicated speed from the speedometer can not be seen as a very accurate measurement, although it will never indicate a speed lower than the true speed.

1.3.2 Satellite Positioning Systems

There exists several satellite positioning systems, including the U.S. developed Global Positioning System (GPS) and its Russian counterpart, the Global Navigation Satellite System (GLONASS). Common for all these, are that they use earth-orbiting satellites to determine the position and/or speed of a receiver on the ground. The receiver can measure the distance between it and the satellite by measuring the time a signal uses to reach the receiver.

Typically, four satellites must be visible from the receiver, three to triangulate the position by measuring the distance to and position of the satellites, and a fourth to correct for clock error in the receiver [6].

Accuracy GPS provides two different services, the Precise Positioning System (PPS) intended for military use and only available to authorized users, and the Standard Positioning Service (SPS) which is available to all GPS users. The accuracy of GPS speed measurements using the SPS has been reported to be as low as 0.05 m/s with a confidence of 99.9 %, and can therefore be seen as an accurate measure of the true speed [7].

1.4 Doppler Radar Speed Measurement - Previous Approaches

The use of the Doppler effect to measure relative speed is well known, and several approaches to speed measurement by radar systems have been proposed and realized. However, most concentrate on the measurement of the speed of a vehicle from a static point, e.g. police speed measurement radars, where the vehicle approaches the point of measurement. But a few publications describe on-board radar speed-measurement systems.

1.4.1 Doppler Radar Speed Measurement 1

A 1978 paper describes a speed measurement system for use by agricultural tractors based on a single 10.525 GHz Doppler radar sensor [8]. The system was tested a tractor running at a constant speed, using a frequency counter with 10 s integration time to find the average Doppler frequency.

Accuracy Two different radar types were tested. An accuracy of 0.5 % was reported when the first type was used, and 2.0 % for the other, compared to a fifth wheel assembly connected to the front wheel of the tractor.

1.4.2 Doppler Radar Speed Measurement 2

A U.S. patent from 1991 presents a simple Doppler radar measurement system where the speed is measured by comparing the change in the Doppler frequency from one measurement to the next [9]. Nothing is stated about the accuracy of this system.

1.4.3 Doppler Radar Speed Measurement 2

The publications by W. Kleinhempel et al. propose a system based on a 61 GHz radar module where two modules set up in a janus configuration (see Figure 1) are used to measure vehicle, ground speed [1, 10, 11]. The Janus configuration is used to eliminate the impact of the ever-changing pitch angles of the vehicle and the system exploits the Doppler effect to measure the speed, as will be described in Section 2.2. A measurement algorithm based on correlating the received Doppler spectra with pre-estimated spectra of known speeds was used to determine the speed of the vehicle. Techniques such as down-converting the relevant part of the spectra and changing the sampling frequency and length of the FFT transform was used improve the measurements as much as possible. This system is the best documented earlier approach to Doppler radar speed

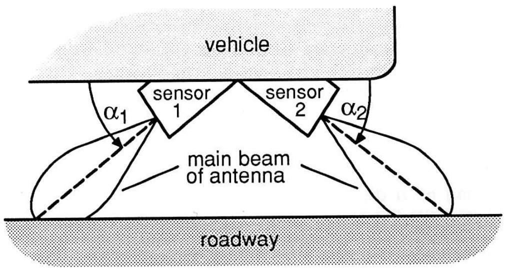


Figure 1: Janus configuration of Doppler radar sensors [1]

measurement that was found. A system very similar to this is described in a 1993 U.S. patent by U.S. Philips Corporation [12].

Accuracy This approach reported a standard deviation (σ) of the measurement errors in the range of 0.6 % compared to a Peiseler wheel mounted on the side of the vehicle, using a measurement interval of 40 ms. Vehicle speeds up to 250 km/h could be measured [1].

1.5 The Proposed Solution

This section will present an outline of the speed measurement system that this project aims to design. A simple block diagram is presented in Figure 2. The system will be mounted on a suitable spot on the vehicle. As the problem

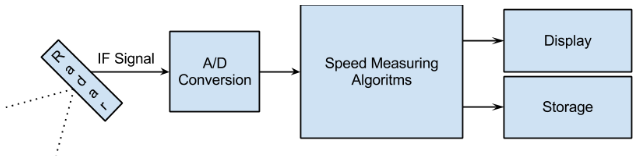


Figure 2: High-level block diagram of the proposed system

text states, the hardware was available at the time the project started. The radar is a 24.100 GHz module from Microwave Solutions Inc., and the signal processor is the ARM Cortex M4 processor incorporated on a microcontroller chip produced by NXP Semiconductors.

The system will consist of a single Doppler radar module, so a Janus configuration as described in Section 1.4.3 will not be possible. However, the algorithm type used there, based on correlating the Doppler- and estimated spectra, stands out as a promising method to test in this project. A simpler method, using only the maximum value of the Doppler spectra itself to measure the speed will also be tested.

1.6 Thesis Outline

The following points will briefly describe the contents of the different sections in the rest of this thesis.

- Section 2 will start by describing some of the relevant and necessary theory behind the Doppler radar and the signal processing used to perform the speed measurements. It will also describe the process of theoretically estimating the received Doppler frequency spectrum.
- Section 3 will describe the hardware that were made available by Q-Free ASA.
- Section 4 describes the process of digitizing the intermediate-frequency signal that is received from the Doppler radar sensor.
- Section 5 will present the speed measuring algorithms.
- Section 6 will then describe how the software was implemented, including implementation of the speed measuring algorithms.

- Section 7 presents how the system and algorithms were tested.
- Section 9 will then present the results obtained during the tests.
- Section 10 will discuss the results, comparing the different test cases and algorithms.
- Section 11 summarizes the results and presents some future work that can be done to improve the system.
- Appendix A includes the source code for the system.
- Appendix B includes Matlab scripts used for simulations, to estimate Doppler power spectra, and to plot the results.

2 Theoretical Background

This section will present some of the necessary theoretical background, starting with a description of the Doppler radar, continuing with the theory behind theoretically estimating the received Doppler spectra, and finishing with some relevant digital signal processing theory.

2.1 Radar

Radar, or *Ra*dio *d*etection *a*nd *r*anging, has been widely used since the 1930's, and the technology dates back to Heinrich Hertz's experiments with Maxwell's theories in 1886 [13]. As early as 1904, a German engineer named Christian Hülsmeier obtained a patent entitled "Hertzian-wave Projecting and Receiving Apparatus Adapted to Indicate or Give Warning of the Presence of a Metallic Body, Such as a Ship or a Train, in the Line of Projection of Such Waves" [14].

In 1922, a wooden ship was detected by engineers at the U.S. Naval Research Laboratory (NRL) by using a CW wave-interference radar, and the first detection of an aircraft was made in 1930 [13]. The attack on Pearl Harbor, Hawaii, was detected by long-range early warning pulse radars developed by the U.S. Army Signal Corps. The British independently developed similar systems, and the so-called Chain Home radar stations deployed on the East and South Coasts of Britain was in operation from 1938 until the end of World War II [13].

After the war, the development of radar continued, and today, radar is used not only for military purposes. Applications of radar includes Air Traffic Control, air and sea navigation and safety, meteorological surveillance and law enforcement [13].

2.1.1 The Radar Equation

The power radiated by a radar transmitter is P_t. When this power is radiated out from an isotropic antenna (uniformly in all directions), the power density at a range R from the antenna is given by

$$P_d = \frac{P_t}{4\pi R^2} \tag{2.1}$$

Radar systems use directional antennas to concentrate the transmitted power P_t in a desired direction. The increase in power compared to an isotropic antenna in this direction is the *gain* of the antenna, G [13]. This gain is expressed in dBi, or *dB compared to an isotropic antenna.* The power density can now be written as

$$P_d = \frac{P_t G}{4\pi R^2} \tag{2.2}$$

When the energy described in Equation (2.2) hits an object, most of it is absorbed and scattered, but some of it is reflected back in the direction of the radar. Objects of different materials, sizes and shapes will reflect different amounts of power and this amount is denoted as the *radar cross section*, σ, of the object. The radar cross section is measured in area (m^2) as seen by the radar. For an arbitrary object, this area is expressed as the area of a flat metal reflector which would reflect the same amount of power back to the radar [15].

When this is combined with Equation. (2.2), the reflected power from the target becomes

$$P_{rf} = \frac{P_t G}{4\pi R^2} \quad \sigma \tag{2.3}$$

This power is reflected in all directions and the power density of the reflected radiation at the radar source is therefore [15]

$$P_d = \frac{GP_t}{4\pi R^2}\frac{\sigma}{4\pi R^2} = \frac{P_t G\sigma}{(4\pi)^2 R^4} \tag{2.4}$$

If the effective area of the receiving antenna is A_e, the power received by the radar is given by

$$P_r = \frac{P_t \sigma G A_e}{(4\pi)^2 R^4} \tag{2.5}$$

The relationship between the area of the receiving antenna and its gain can be described as [13]

$$G = \frac{4\pi A_e}{\lambda^2} \tag{2.6}$$

and since most radars (including the module used in this project) use the same antenna for transmission and reception, this can be substituted into Equation (2.5) and the received power becomes

$$P_r = \frac{P_t G^2 \lambda^2 \sigma}{(4\pi)^3 R^4} \tag{2.7}$$

2.2 The Doppler Effect

When a source transmitting an electromagnetic (or any other) wave is moving relative to an observer, a frequency shift between the transmitted signal and the received signal will be observed. This effect is known as the Doppler effect and the frequency change is known as the Doppler shift or the Doppler frequency.

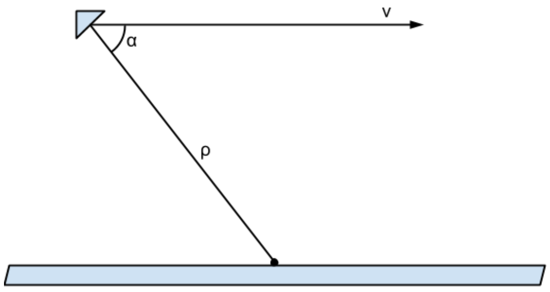


Figure 3: Orientation of a Doppler radar with respect to the surface

Consider a system where the transmitter and receiver is in the same position, and the system is moving relative to some surface (see Figure 3). The transmitted and received signals can be represented as [16, 17]

$$E_T = A \sin 2\pi f t \tag{2.8}$$

$$E_R = B \sin \left[2\pi f t - \varphi\right] \tag{2.9}$$

where A and B are constants and φ is the phase shift due to the propagation delay time. This phase shift can be described as

$$\varphi = 2\pi \frac{2\rho}{\lambda} \tag{2.10}$$

where ρ is the distance from the system to the point of reflection and λ is the wavelength of the transmitted signal. When the system is moving relative to the point of reflection, ρ, and hence φ, varies with time. Over a short time-period ρ may be represented by [16]

$$\rho(t) = \rho_0 - vt \cos \alpha \tag{2.11}$$

where ρ_0 is the value of ρ at $t = 0$ and α is the angle between the direction of the velocity v and the direction towards to the point of reflection. Now, Equation (2.9) can be written as

$$\begin{aligned} E_R &= B \sin \left[2\pi f t - \frac{4\pi}{\lambda}\left(\rho_0 - vt \cos \alpha\right)\right] \\ &= B \sin \left[\left(2\pi f + \frac{4\pi v \cos \alpha}{\lambda}\right) t - \varphi_0\right] \end{aligned} \tag{2.12}$$

where $\varphi_0 = \frac{4\pi\rho_0}{\lambda}$ is a fixed and insignificant phase lag. The more important fact seen here is that the received signal differs from the transmitted signal by a time varying term, i.e. a frequency:

$$f_R = f + \frac{2v}{\lambda} \cos \alpha \tag{2.13}$$

The Doppler frequency shift f_d is therefore:

$$f_d = f_R - f = \frac{2v}{\lambda} \cos\alpha \tag{2.14}$$

When the target is closing (when v is positive), the sign of the Doppler frequency is positive. Similarly, the sign of the Doppler frequency is negative when the target is receding.

2.2.1 The Doppler Spectrum

The Doppler frequency equation (2.14) gives the Doppler frequency for one single angle α, but in reality, the radar radiates out inn all directions, with a peak power in the direction towards the target. Hence, the received Doppler frequencies compose a spectrum of many different frequencies. As will be shown graphically when estimating the Doppler frequency spectrum, the width of the spectral peak is not only dependent of the radiation pattern of the radar, but directly proportional to v:

$$\Delta f = f_{dl} - f_{du} = \frac{2v}{\lambda}(\cos\alpha_l - \cos\alpha_u) \tag{2.15}$$

where α_l and α_u are some lower and upper angles, e.g. the 3 dB point of the antenna pattern [1].

2.3 The CW Doppler Radar

The CW, or continuous-wave, radar transmits an unmodulated and continuous signal of frequency f, and the portion of the signal that is reflected by the target is received and compared to the output signal.

When the target is moving relative to the radar, the received signal is shifted in frequency as given by Equation (2.14). As shown in Figure 4, an oscillator generates a stable signal with a given amplitude and frequency. This signal is filtered and divided into to approximately equal signals [15]. One of the signals is radiated by the transmit antenna, and the other is used as a reference signal to a balanced mixer. The mixer compares the received signal with the transmitted signal and produces an intermediate frequency (IF) signal with the Doppler frequency f_d.

The received signal will be on the form $f \pm f_d$, but the sign will be lost in the process, and only the positive Doppler frequency will be detected. Hence, the unmodulated CW Doppler radar will only detect that the presence and magnitude of a relative motion, and not its direction [13].

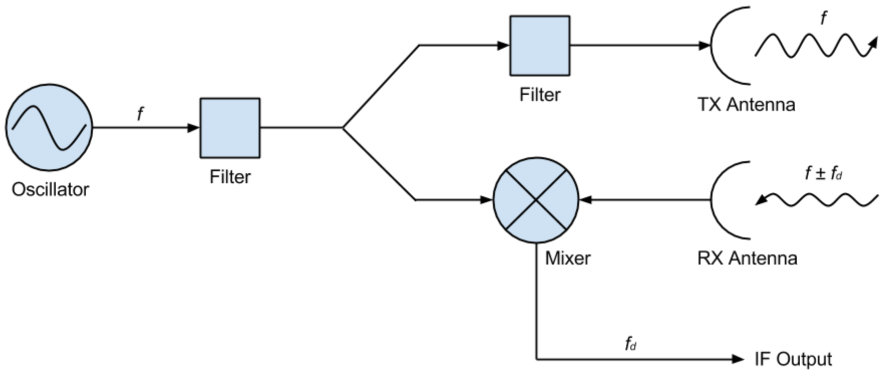


Figure 4: CW Doppler radar block diagram

2.4 Theoretical Received Power Spectrum

The Doppler signal received from the radar is not a single frequency, and to be able to better determine the velocity of the vehicle, a method for estimating the received power spectrum of the different Doppler frequencies is desired. This estimate can be used to detect the velocity by correlating the measured spectrum with estimated spectra for different velocities, as will be shown in Section 5.3.

A relatively simple estimate of the Doppler spectrum can be found when expanding the radar equation to two or three dimensions. As was shown by [1], expanding to two dimensions by taking into account the vertical plan, yields a good estimate of the true spectrum.

2.4.1 The Radar Equation Expanded to Two Dimensions

The radar equation derived in Section 2.1.1 is valid only in one direction from the radar to a target, i.e. at constant angles both in the horizontal and vertical planes, with the radar pointing directly at the target. This is sufficient for estimating the maximum received power and range, but not for estimating the complete received power spectrum.

When adding the vertical dimension, the gain G, the radar cross section σ, and the distance R, all varies with the inclination angle α (see Figure 5). The gain also depends on the tilt, θ, of the radar sensor (45° in the figure). When inserting these factors in the radar equation, an expression of the power of the

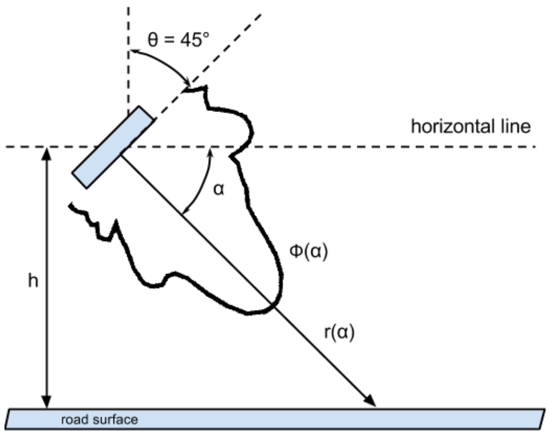


Figure 5: The geometry of the radar setup in two dimensions

received signal as a function of the inclination angle can be written as [1]

$$P_d(\alpha) = c_0 \frac{\sigma(\alpha)\phi^2(\alpha)}{r^4(\alpha)} \tag{2.16}$$

where $c_0 = \frac{P_t\lambda^2}{(4\pi)^3}$ is the constants of the radar equation, $\sigma(\alpha)$ is the radar cross section of the target, $\phi(\alpha)$ is the antenna gain, and $r(\alpha) = \frac{h}{sin(\alpha)}$ is the distance from the radar to a point on the target.

2.4.2 Radar Cross Section of Asphalt ($\sigma(\alpha)$)

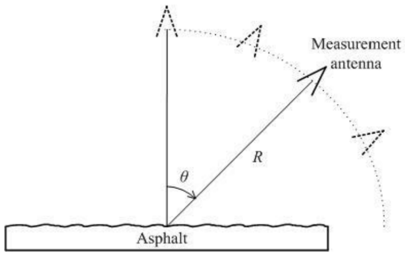
The radar cross section as described in Section 2.1.1 can be further expressed as

$$\sigma = \sigma_0 A_0 \tag{2.17}$$

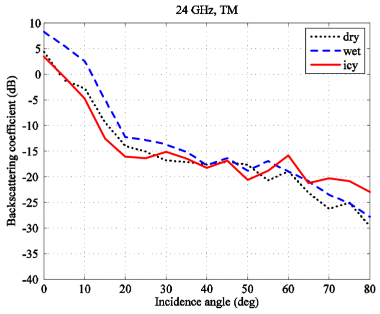
where σ_0 is the back-scattering coefficient of the target (in dB) and A_0 is the illuminated area [18, 19].

Measurements done by [18] has shown the back-scattering coefficient for a 24 GHz radar to be as shown in Figure 6b for different incidence angles. These results will be used for $\sigma_0(\alpha)$ in the power spectrum estimate.

The area A_0 can be approximated by calculating the area within the 3 dB beam-width of the antenna beam that illuminates the target [18], see Figure 7.



(a) Incidence angle used in Fig. 6b related to the surface [18]



(b) Back-scattering coeff. at 24 GHz [18]

Figure 6: Measured back-scattering coefficient for asphalt roads.

This area varies with the radar tilt θ, and can be written as:

$$A_{0d}(\theta) = \frac{1}{h}\tan\left(\theta - \frac{\theta_{3dBv}}{2}\right) - \frac{1}{h}\tan\left(\theta + \frac{\theta_{3dBv}}{2}\right) \tag{2.18}$$

$$A_{0w}(\theta) = \frac{h}{\sin(\theta)}\cdot\tan\left(\frac{\theta_{3dBh}}{2}\right)\cdot 2 \tag{2.19}$$

$$A_0(\theta) = A_{0d}\cdot A_{0w} \tag{2.20}$$

where A_{0d} is the depth of the illuminated field, A_{0w} is the width of the field, θ_{3dBv} is the vertical 3 dB beam-width, and θ_{3dBh} is the horizontal 3 dB beam-width.

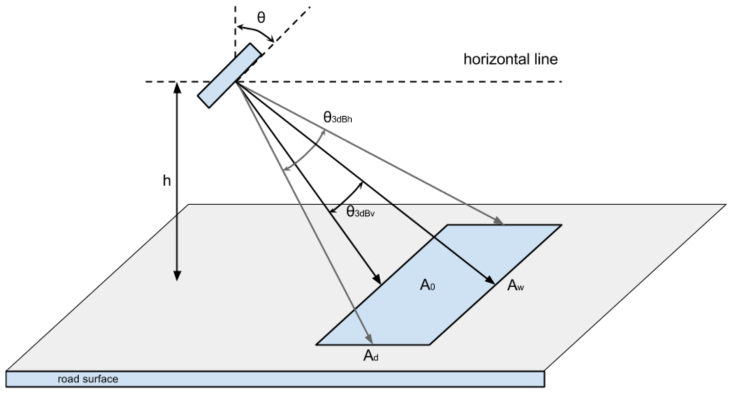


Figure 7: The geometry of radar cross section area

The area also varies with α, but this is left out of this approximation.

2.4.3 Antenna Gain ($\phi(\alpha)$)

The antenna diagram supplied by the radar manufacturer gives the gain at all 360° in the vertical and horizontal planes around the antenna (Figure 8). By measuring the diagram, the resulting curve together with the antenna gain can be used as $\phi(\alpha)$ in Equation (2.16).

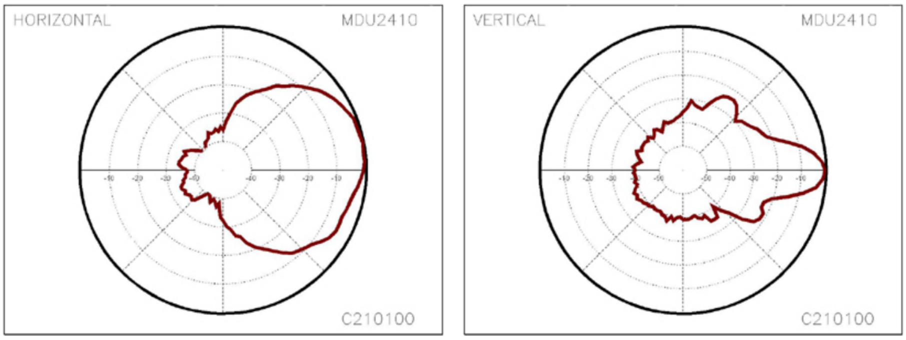


Figure 8: Antenna diagrams from the radar module data sheet.

According to the manufacturer, the gain of the antenna is 10 dBi [20], so this must be added to the value from the antenna diagram, giving 10 dBi gain at the direction of maximum power (0°). A plot of $\phi(\alpha)$ from $-90°$ to $+90°$ in the vertical plane can be seen in Figure 9.

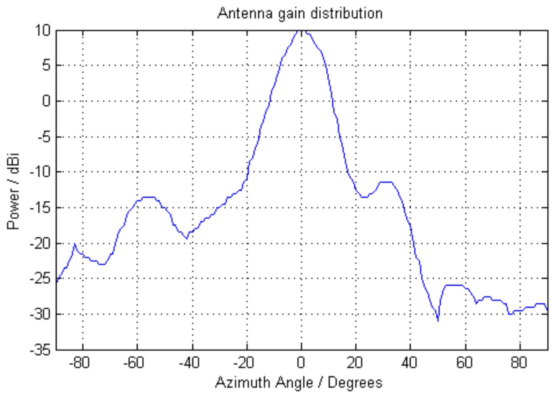
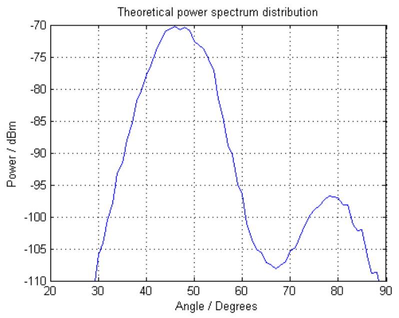


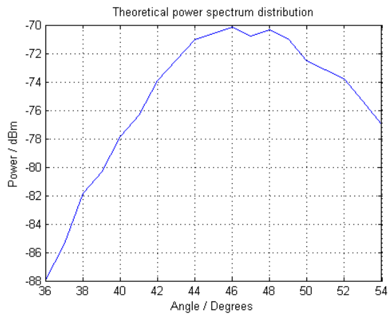
Figure 9: A plot of the antenna gain distribution $\phi(\alpha)$ in the vertical plane

2.4.4 Doppler spectrum estimation for different velocities

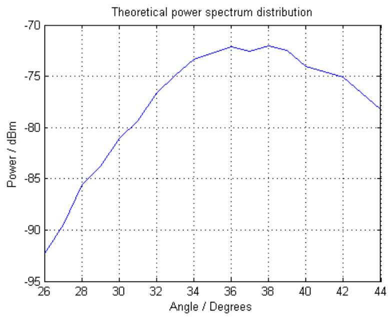
Equation (2.16) gives an estimate of the received power at different angles. Plots of $P_d(\alpha)$ are shown in Figures 10a,10b and 10c.



(a) $P_d(\alpha)$ when $\theta = 45°$



(b) Zoomed-in view of figure 10a



(c) Zoomed-in view when $\theta = 35°$

Figure 10: Estimates of received power spectra. 0° is along the horizontal line.

In these figures, one can see how the received spectrum is estimated to be, and how the power drops when the tilt angle is lowered. But this does not show how the Doppler spectra changes with different velocities. Hence, an estimate of the different Doppler frequencies at different velocities is desired.

To calculate the velocity from the received signal, the Doppler equation as described in Section 2.2 is used. When Equation (2.14) is inverted, α can be given as a function of f_d and v_0, where f_d is the Doppler frequency, and v_0 is the velocity of the vehicle [1]:

$$\alpha = \arccos(\frac{f_d \lambda}{2v_0}) \tag{2.21}$$

This can be inserted into Equation (2.16) to produce an estimate of the Doppler spectral distribution $P_d(f_d, v_0)$. When choosing a velocity v_0, the spectrum can be plotted for the different frequencies:

$$P_d(f_d, v_0) = c_0 \frac{\sigma(\arccos(\frac{f_d \lambda}{2v_0}))\phi^2(\arccos(\frac{f_d \lambda}{2v_0}))}{r^4(\arccos(\frac{f_d \lambda}{2v_0}))} \tag{2.22}$$

2.4.5 Some Estimations

Using Equation (2.22), some estimations with different parameters will be shown. The parameters obtained from the radar module data sheet are as follows [20]:

- $P_t = 7$ dBm $= 5$ mW
- $G = 10$ dBi
- $\lambda = 0.0124$ m
- $\theta_{3dBv} = 18°$
- $\theta_{3dBh} = 72°$

With these parameters, the constant c_0 becomes: $c_0 = \frac{5 \times 10^{-3} \times 0.0124^2}{(4\pi)^3} = 2.55 \times 10^{-6}$. All estimations use $h = 0.5$m (assuming the radar is placed on the rear bumper of the vehicle), and the spectra are estimated for 20, 40, 80 and 160 km/h. Figure 11 show plots of the estimations, where the radar tilt θ is 45°.

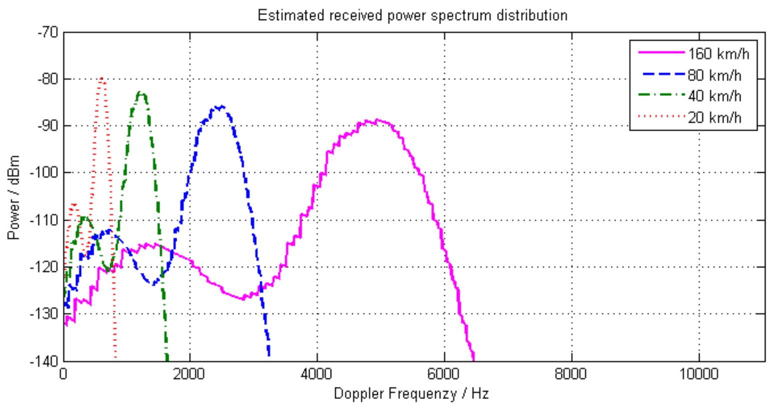


Figure 11: Estimated power spectra for three different speeds when $\theta = 45°$

These estimations clearly show how the with of the spectrum is proportional to v_0, as shown in Section 2.2.1. The rugged structure of the estimations is due to the fact that the antenna gain values used has a resolution of 1°, so rounding was used to find the closest value during the estimation.

2.5 Digital Signal Processing Theory

This section will give a brief introduction to some of the digital signal processing techniques used in the project.

2.5.1 The Fast Fourier Transform (FFT)

The Fourier Transform is the transformation of a signal from the time-domain to the frequency-domain. The following gives a very brief introduction to the Fast Fourier Transform algorithm used to effectively calculate the Discrete Fourier Transform.

Let $x(n)$ be a finite-duration discrete-time sequence of length L (i.e. $x(n) = 0$ outside the range $0 \leq n \leq L - 1$). This sequence has a Fourier transform described as

$$X(\omega) = \sum_{n=0}^{L-1} x(n)e^{-j\omega n}, \qquad 0 \leq \omega \leq 2\pi \qquad [2] \tag{2.23}$$

The *Discrete Fourier Transform* (DFT) is found by evaluating this Fourier transform at a set of N equally spaced frequencies $\omega_k = 2\pi k/N, k = 0, 1, 2, \ldots, N-1$, where $N \geq L$, essentially sampling the frequency domain of the signal. This gives a DFT with the samples

$$X(k) = X(\frac{2\pi k}{N}) = \sum_{n=0}^{L-1} x(n)e^{-j2\pi kn/N}$$

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j2\pi kn/N}, \qquad k = 0, 1, 2, \ldots, N-1 \qquad [2] \tag{2.24}$$

where the upper index is increased from $L - 1$ to $N - 1$ since $x(n) = 0$ for $n \geq L$.

The sequence $x(n)$ can be recovered from X(k) by the inverse DFT (IDFT) formula:

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k)e^{j2\pi kn/N}, \qquad n = 0, 1, 2, \ldots, N-1 \qquad [2] \tag{2.25}$$

The direct calculation of the DFT requires N^2 complex multiplications [2]. Hence, it is clear that the workload of directly calculating the DFT becomes massive when N increases, with 4.194.304 complex multiplications at $N = 2048$. However, when N is selected to be a number 2^v, the calculations can be divided up into smaller and smaller elements, all the way down to a 2-point DFT (see Figure 12). This algorithm is called the radix-2 FFT (*Fast Fourier Transform*) and a detailed explanation of it can be found in many digital signal processing textbooks, including [2]. In short, the divide-and-conquer approach

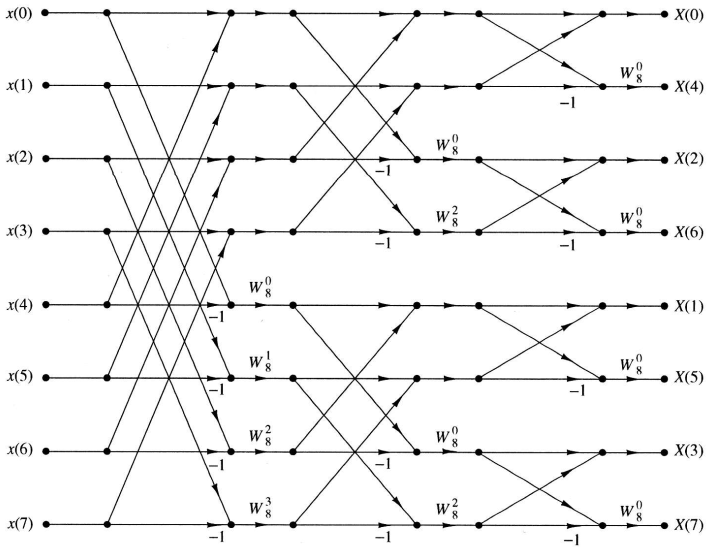


Figure 12: Eight-point FFT Algorithm [2]

and other techniques used in the FFT algorithm reduces the number of complex calculations to $(N/2)\log_2 N$. This gives just 11.264 complex calculations at $N = 2048$, a significant improvement compared to the direct calculation, making the radix-2 algorithm the most widely used FFT algorithm [2], and suitable for digital signal processors. Other basic building blocks, such as a 4-point DFT, can be used, e.g. to construct a radix-4 type FFT.

2.5.2 The Power Density Spectrum (Periodogram)

A finite-energy deterministic signal $x_a(t)$ has a total energy of

$$E = \int_{-\infty}^{\infty} |x_a(t)|^2 dt = \int_{-\infty}^{\infty} |X(F)|^2 dF \tag{2.26}$$

where $X(F)$ is the Fourier transform of $x_a(t)$, and $|X(F)|^2$ represents the signal's energy distribution [2]. This quantity can also represented by the symbol $S_{xx}(F)$, that can be shown to be the Fourier transform of the auto-correlation of $x_a(t)$. The energy density spectrum of a sampled version $x(n)$ of the signal, will be

$$S_{xx}(f) = |X(f)|^2 \tag{2.27}$$

However, many natural signals are best characterized statistically as random processes, including the IF signal from the radar sensor. Such signals does not have finite energy and does not possess a Fourier Transform. Hence, the energy density spectrum cannot be calculated. But these signals have finite average power and can be characterized by a power density spectrum [2].

The power density spectrum can be estimated to form a Power Density Estimate, or periodogram. This thesis will not go into the details of the derivation of this estimate, but it includes using an estimate for the auto-correlation function to calculate the power density estimate $P_{xx}(f)$ from samples of a realization of the random process. This power density estimate can be expressed as

$$P_{xx}(f) = \frac{1}{N}\left|\sum_{n=0}^{N-1} x(n)e^{-j2\pi fn}\right|^2 = \frac{1}{N}|X(f)|^2 \qquad [2] \tag{2.28}$$

where N is the number of samples.

The variance of the periodogram does not converge to zero as $N \to \infty$. In fact, is can be shown that the variance is

$$\operatorname{var}\left[P_{xx}(f)\right] = \Gamma_{xx}^2(f)\left[1 + \left(\frac{\sin 2\pi fN}{N\sin 2\pi f}\right)^2\right] \tag{2.29}$$

$$\lim_{N\to\infty} \operatorname{var}\left[P_{xx}(f)\right] = \Gamma_{xx}^2(f) \tag{2.30}$$

where Γ_{xx} is the true power density spectrum of the process. Hence, the periodogram is not a consistent estimate, as it does not converge to the true power density spectrum [2]. Several methods have been developed to reduce the variance, and some of them will be presented in the following sections.

2.5.3 The Barlett and Welch Methods for Power Spectrum Estimation

The Barlett and Welch methods are non-parametric methods for power spectrum estimations, i.e. they make no assumptions about the process that generated the data. Both methods apply techniques to reduce the variance of the estimation, at the cost of reducing the spectral resolution.

Barlett's method is to average multiple periodograms by dividing the N-point sequence into K non-overlapping segments of length M. The K periodograms are calculated and averaged to obtain the Barlett power spectrum estimate:

$$P_{xx}^{B}(f) = \frac{1}{K}\sum_{i=0}^{K-1} P_{xx}^{(i)}(f) \qquad [2] \tag{2.31}$$

This reduces the variance of the estimate with a factor K, and the frequency resolution is reduced by the same factor [2].

The Welch method introduces two modifications to the Barlett method: The K segments are allowed to overlap, and the data segments are windowed prior to calculating the periodogram. If there is zero overlap, $L = K$ segments are obtained, if there is 50% overlap, $L = 2K - 1$ segments are obtained, and so on. The windowed, or *modified*, periodogram can now be described as [2, 21]

$$\tilde{P}_{xx}^{(i)}(f) = \frac{1}{MU}\left|\sum_{n=0}^{M-1} x_i(n)w(n)e^{-j2\pi fn}\right|^2, \qquad i = 0, 1, \ldots, L-1 \tag{2.32}$$

where $w(n)$ is the window function and U is a normalization factor for the power in this function such that

$$U = \frac{1}{M}\sum_{n=0}^{M-1} w^2(n) \qquad [21] \tag{2.33}$$

When these modified periodograms are averaged, we get the Welch power spectrum estimate

$$P_{xx}^{W}(f) = \frac{1}{L}\sum_{i=0}^{L-1} (\tilde{P})_{xx}^{(i)}(f) \qquad [21] \tag{2.34}$$

If a triangular window (Barlett window) is used, the variance is reduced by a factor of L if there is zero overlap, and a factor of $\frac{8}{9}L$ if there is 50% overlap. [2, 21].

Both the Barlett and Welch methods yields consistent estimates for the power density spectrum, as the variance converges to zero when $N \to \infty$, and M and K are allowed to grow with N [2].

2.5.4 Comparing Power Density Spectra

The mathematical operation of correlation is used to measure the similarity of two signals. The operation is closely related to convolution, and can be expressed as

$$r_{xy}(l) = \sum_{n=-\infty}^{\infty} x(n)y(n-l), \qquad l = 0, \pm 1, \pm 2, \ldots \qquad [2] \tag{2.35}$$

The result is a new signal, where l represents a time lag, and the value of $r_{xy}(l)$ represents the similarity between the signals at the relative lag l. When $x(n) = y(n)$, $r_{xy}(l)$ is known as the auto-correlation sequence, with a maximum value at $l = 0$, meaning that the signal matches best with itself at zero lag [2]. When $x(n) \neq y(n)$, $r_{xy}(l)$ is known as the cross-correlation sequence, and l represents the time lag at which the signals match best.

As can be shown, correlation in the time-domain corresponds to a multiplication in the frequency domain. If the Fourier-transform of the correlation sequence is $S_{xy}(f)$, and the Fourier-transforms of $x(n)$ and $y(n)$ are $X(\omega)$ and $Y(\omega)$, the correlation can be expressed as [2]:

$$\begin{aligned} S_{xy}(\omega) &= \sum_{l=-\infty}^{\infty} r_{xy}(l) = \sum_{l=-\infty}^{\infty} \left[\sum_{n=-\infty}^{\infty} x(n)y(n-l) \right] e^{-j\omega l} \\ &= X(\omega)Y(-\omega) \\ &= X(\omega)Y^*(\omega), \qquad \text{if } y(n) \text{ is real} \end{aligned} \tag{2.36}$$

Unfortunately, the Fourier-transforms of the signals that must be compared in one of the speed measuring algorithms are not available. However, the estimated Power Density Spectra are available, and a similar approach will be taken when comparing these:

When multiplying the frequency components of the two power spectra, components with high values in both spectra will yield a component with a high value in the correlation spectrum. Similarly, components with low values in one or both of the spectra will yield a relatively low valued component in the correlation spectrum. Hence, the function

$$P_{xy} = P_{xx} \cdot P_{yy}^* \tag{2.37}$$

will give a representation of the "likeness" of the two power spectra P_{xx} and P_{yy}.

3 Available Hardware

This section will describe the hardware equipment that were made available by Q-Free ASA and used to implement the radar measuring system.

3.1 The MDU2410 Doppler Radar Module

The Doppler radar module provided for use in this project is the MDU2410 module from Microwave Solutions Ltd. This module is a K-band (24.100 GHz) motion detector unit that uses the principle of the continuous-wave Doppler radar as described in Section 2.3. The module outputs the Doppler frequency shift f_d as an analog signal that can be amplified and interpreted by signal processing techniques. A detailed description can be found in the official data sheet [20], and the most important characteristics are shown in Table 1:

Item	Symbol	Value
Output frequency	f	24.100 GHz
Wavelength	λ	0.0124 m
Antenna gain	G	10 dBi
Power output	P_t	7 dBm (5 mW)
Equivalent Isotropically Radiated Power	EIRP	17 dBm (min.)
Receiver sensitivity	P_s	-76 dBm (25 pW)
Horizontal 3 dB beam width	θ_{3dBh}	72°
Vertical 3 dB beam width	θ_{3dBv}	18°

Table 1: MDU2410 Radar Specifications

Although the module is intended for indoor motion detection, the specifications implies that it should also be suitable for a velocity-measurement system like the one implemented in this project, where the distance from the radar to the target (road) is smaller than the distance from the ceiling to the floor in an average room. By using the above characteristics and Equation (2.7), the maximum theoretical range of the radar module can be calculated as:

$$R_{\max} = \left[\frac{P_t G^2 \lambda^2 \sigma}{(4\pi)^3 P_s}\right]^{\frac{1}{4}} = \left[\frac{5 \times 10^{-3} \cdot 10^2 \cdot 0.0124^2}{(4\pi)^3 \cdot 2.5 \times 10^{-11}}\right]^{\frac{1}{4}} = 6.27\text{m} \tag{3.1}$$

where σ is set to 1, and P_s is the maximum sensitivity of the radar. This is, of course, a theoretical maximum range, as the radar cross section σ will greatly affect this range.

3.1.1 The Intermediate-Frequency signal

As shown in Section 2.2.1, the Intermediate-Frequency (IF) signal will be composed of a spectrum of all the received Doppler shift frequencies. This section will briefly describe some of the properties of this signal.

Frequencies and Bandwidth

The lowest possible frequency that can be received will always be 0 Hz, corresponding to no relative velocity, i.e. the vehicle is not moving. The highest frequency of interest corresponds to the highest speed that the system should be able to measure. From the specifications, this speed is set to 250 km/h, or approximately 70 m/s. The tilt of the radar, θ, will affect the relevant bandwidth, but for the purpose of this calculation it is set to $45° = \frac{\pi}{4}$ rad. Using the Doppler equation from (2.14), the frequency corresponding to 250 km/h can be calculated:

$$f_{\text{peak}} = \frac{2v}{\lambda}\cos\theta = \frac{2\cdot 70}{0.0124}\cdot\cos\frac{\pi}{4} \approx 8000 \text{ Hz} \tag{3.2}$$

When the radar is tilted 45° the peak of the Doppler spectrum should therefore be at around 8 kHz. But when looking at the radar beam pattern in Section 2.4.3, it is evident that the radar radiates in all directions. Thus, the maximum possible received Doppler frequency at a relative velocity of 250 km/h will be at $\theta = 0°$:

$$f_{\text{max}} = \frac{2v}{\lambda}\cos\theta = \frac{2\cdot 70}{0.0124}\cdot\cos 0 \approx 11300 \text{ Hz}, \tag{3.3}$$

corresponding to a speed of 358 km/h.

The relevant bandwidth of the signal, B, is therefore $\sim$ 11300 Hz, when the radar tilt $\theta = 45°$. Section 6.6.1 will show that a sampling frequency of 22050 Hz was selected. Hence, the final bandwidth becomes 11025 Hz.

DC Component

A DC level of $< \pm150$mV will be present in the IF signal when the radar is operating [15]. This is a sum of all the reflected signals from the radar module itself and static objects in its surroundings. The DC signal will also vary as a function of the ambient temperature.

Noise

After the initial data recording tests were completed, a significant amount of noise was found to be present, especially in the form of what appears to be pink (or 1/F) noise at the lower frequencies. Figure 13 shows a plot of the frequency components of the signal when the vehicle is stationary.

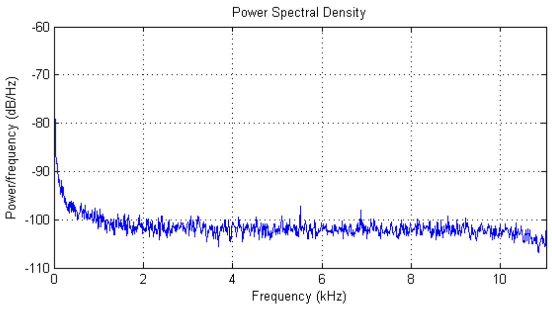


Figure 13: Plot of the received power spectrum when the vehicle is stationary

Amplification
The IF signal is relatively weak, and to apply an amplification of 70dB is described as typical by the radar module manufacturer for a 10 GHz module [15].

3.2 Signal Processing Hardware

In addition to the radar sensor, a hardware platform based on the NXP LPC4357 chip was provided. This is a high-end microcontroller containing both an ARM Cortex M4 processor and a Cortex M0 co-processor, as well as many peripheral units for communication, including USB, networking, display driver, etc. The Cortex M4 core contains functionality that is especially useful for digital signal processing (DSP) applications, such as dedicated floating-point hardware and special SIMD (Single Instruction Multiple Data) instructions [22].

3.2.1 Development Kit

To develop and test the speed measurement application, the MCB4357 evaluation kit from ARM Keil containing the NXP LPC4357 chip was used. The use of such a development kit greatly simplifies the development process, as all the required hardware is placed on the same circuit board. Listed below are some of the most important features of the development board. The full specifications can be found at the Keil website [23].

- NXP LPC4357 Micro-controller, capable of running at frequencies up to 204 MHz and containing up to 136 kB SRAM (72 kB local)
- On-Board Memory: 16MB NOR Flash, 4MB Quad-SPI Flash, 16 MB SDRAM, 16KB EEPROM
- Color QVGA TFT LCD with touch-screen

- MicroSD Card Interface
- Audio CODEC with Line-In/Out and Microphone connector
- JTAG Debug Interface

The microSD card interface is essential for storing the captured data during development and testing of the system, and the audio CODEC (Coder/Decoder) incorporates all the necessary functions for sampling and filtering the analog intermediate frequency signal from the radar. A photograph of the board can be seen in Figure 14.

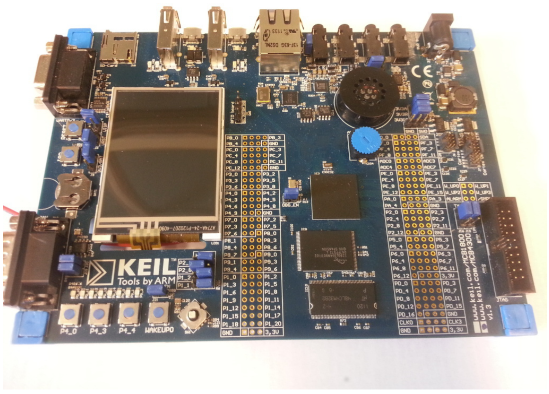


Figure 14: The Keil MCB4357 evaluation kit

3.2.2 Block Diagram of the Hardware Setup

This section will give an overview of the different hardware modules used to capture, analyze and present the speed measurements. A block diagram of the setup can be seen in Figure 15. All of these blocks are part of the MCB4357 development board described in the previous section.

A 3.5mm Jack Connector is used to connect the radar module to the system. The connector is mounted on the board and is connected to the microphone input of the UDA1380 CODEC chip.

The UDA1380 CODEC is used to sample and digitize the analog signal from the radar. It converts the signal into 32 bit two's compliment format, and transmits this to the microcontroller over the I^{2}S bus. The CODEC is further described in Section 4.1.

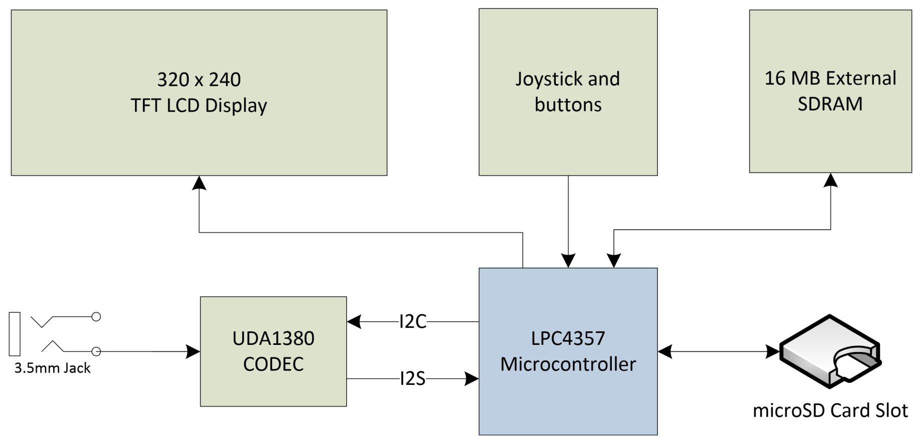


Figure 15: Block Diagram of the Hardware Setup

The LPC4357 microcontroller is used as a Digital Signal Processor and is the core of the system. Here, the digital data is received, signal processing algorithms are applied, and the results are presented using an LCD display. A joystick and buttons are used to enable the user to interact with the system, and external memory is available for storing temporary data. The microcontroller is running a custom made application that will be described in detail in Section 6.

A microSD Card Slot enables the system to store the untreated sampled data. This data can then be used for simulation and algorithm development purposes on a computer, e.g. by using Matlab. In addition, the speed measurements will be stored for future reference.

3.2.3 Power Supply

To minimize the noise generated from the power supply, three Lithium battery-cells were used as the power supply for the complete system. These generated a 10.8 V supply that were regulated to 5 volts using a standard 7805 voltage regulator.

4 Analog to Digital Conversion

As shown in Section 2.3, the radar sensor outputs an intermediate-frequency signal with a frequency that corresponds to the Doppler frequency shift. In reality, this signal will contain a spectrum of Doppler frequencies as described in Section 3.1.1.

This section will describe how the IF signal was converted from analog to digital format.

4.1 Using an Audio Codec to Sample the IF signal

The bandwidth of the IF signal received from the radar module is $\sim$ 11300 Hz as shown in Section 3.1.1, which is well within the audible range of frequencies. Knowing this, one can take advantage of the audio CODEC included on the MCB4357 development board. This audio CODEC chip, the UDA1380 from NXP Semiconductors, contains all the functionality that is needed to filter, amplify and digitize the IF signal [24]. The most relevant specifications are as follows:

- 24-bit data path for Analog-to-Digital Converter (ADC)
- Sample frequencies (f_s) from 8 to 55 kHz
- Mono microphone input with a Low Noise Amplifier (LNA) incorporating 29 dB fixed gain and a Variable Gain Amplifier (VGA) from 0 to 30 dB in steps of 2 dB.
- ADC Signal-to-Noise ratio at $f_s = 48$ kHz: 92 dB (min.), 97 dB (typ.)
- ADC + LNA (Microphone input) Signal-to-Noise ratio at $f_s = 48$ kHz: 85 dB (typ.)
- I^{2}C control interface
- I^{2}S data transfer interface

The CODEC also contains a Digital-to-Analog part, Line in/out and a headphone driver, but none of these are relevant for this application, and will be disabled.

As the specifications show, the CODEC is well suited for sampling the IF signal: The 24 bit A/D Converter ensures high dynamic resolution and low quantification error, a sampling frequency of $\sim$ 22600 Hz is possible, and the microphone input amplifier can provide a total gain of 59 dB. The CODEC can be programmed by using the I^{2}C bus interface. Here, the different components can be enabled or disabled, gain settings can be adjusted, etc.

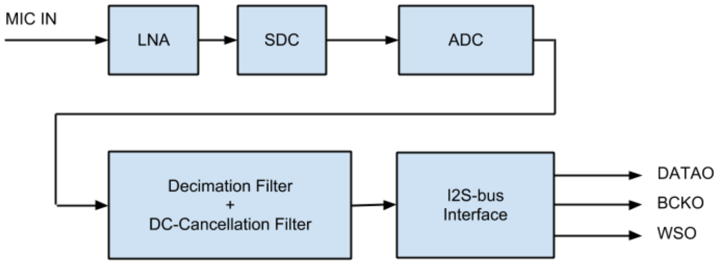


Figure 16: Simple block diagram of the UDA1380 CODEC's ADC Features

Figure 16 gives an overview of the relevant features of the CODEC chip. The signal first enters the Low Noise Amplifier (LNA). Here, the signal is amplified with a fixed gain of 29 dB, in addition to the programmable gain from the Variable Gain Amplifier (VGA) of up to 30 dB. After the LNA, the signal is converted by the Single-Ended to Differential Converter (SDC) to form a differential signal from v_- to v_+ instead of a signal from 0 V to v_{max}. This signal is then fed to the A/D-converter which samples the signal at $128 \cdot f_s$, before the signal is sent through the decimation filter. This filter decimates the signal in two stages: The first realizes a $\frac{\sin x}{x}$ characteristic with a decimation factor of 16. The second stage consists of 3 half-band filters, each decimating by a factor of two [24]. Table 2 shows the filter characteristics of the decimation filter.

Item	Condition	Value (dB)
Pass-band ripple	0 to $0.45 f_s$	0.01
Stop band	$>0.55 f_s$	- 70
Dynamic range	0 to $0.45 f_s$	$>$ 135
Digital output level	at 0 dB input analog	- 1.5

Table 2: Decimation Filter characteristics

In addition to the decimation filter, a DC filter can be used to remove the DC component of the signal. Finally, the signal is converted to the I^{2}S format and transmitted to the micro-controller (see the next section for details).

4.2 Digital Data Transfer over the I^{2}S Bus

The I^{2}S, or Inter-IC Sound, is a serial bus developed by Philips Semiconductors for transferring digital audio data [25]. As shown in Figure 17 [25], the bus consists of three lines: A serial clock (SCK), word select (WS), and serial data

(SD). The bus is organized in a master-slave configuration where the master transmits both the SCK and the WS signals. When the master is transmitting, all lines are driven by the master, and when the master is receiving, the SD line is driven by the slave. The SCK and WS lines can also be shared by the sender and transmitter.

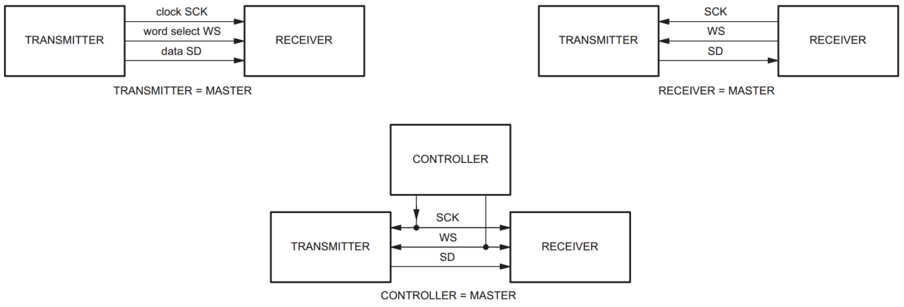


Figure 17: Simple system configurations of the I^{2}S bus

The serial data line transfers 32-bit words of data, where the MSB is transferred first (See Figure 18 [25]). Because the MSB is transferred first, the master and slave can operate with different word lengths: The least significant bits is simply ignored when receiving or set to zero when transferring if the word length is not 32 bits. The data is transferred in two's compliment format.

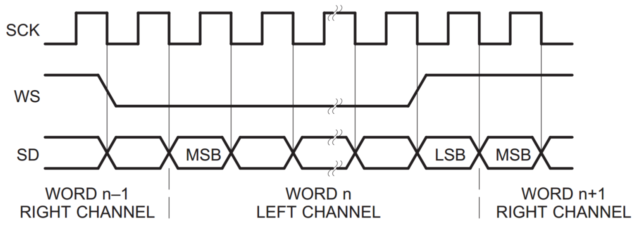


Figure 18: I^{2}S bus timing diagram

The word select line is used to indicate the channel that is being transmitted: When the line is a logical 0, channel 1 (left) is transmitted, when the line is logical 1, channel 2 (right) is transmitted. When the audio is on mono format (only one channel), only the left channel is used. The WS line changes one clock period before the MSB is transmitted on the SD line.

The serial clock needs to cycle once per transferred bit on the SD line, as shown in Figure 18. If the sampling frequency is $22,050$ Hz, the word length is 32 bits, and there are two channels, the frequency of the SCK line must be $f_{SCK} = 22050 \cdot 32 \cdot 2 = 1.411$ MHz.

4.3 IF Signal Filtering

As mentioned in Section 3.1.1, the IF signal contains a low level DC signal and some noise at the lower frequencies. These frequencies must hence be filtered out, as this noise would hinder the speed measurement algorithms, leading to poor speed measurements.

The ARM CMSIS DPS library used for the software development contains functions for filtering signals using a finite impulse response (FIR) filter. Exploiting this, a FIR filter was designed using a trial-and-error approach in Matlab. A high-pass filter cutoff frequency of 3 to 80 Hz is recommended by the radar module manufacturer for a 10 GHz module [15], but different cut-off frequencies and filter orders were tested to get the best possible filter.

After some testing, a filter with a cut-off frequency of 160 Hz and 100 coefficients were selected. Figure 19 shows the spectrum of the radar module before and after filtering, using an input signal where the vehicle is stationary, and Figure 20 show the frequency and phase response of the filter. As can be seen here, a 100th order FIR filter is just enough to lower the frequency components close to 0 Hz to an acceptable level.

The 100 coefficients of the filter can be found in Appendix A.2.

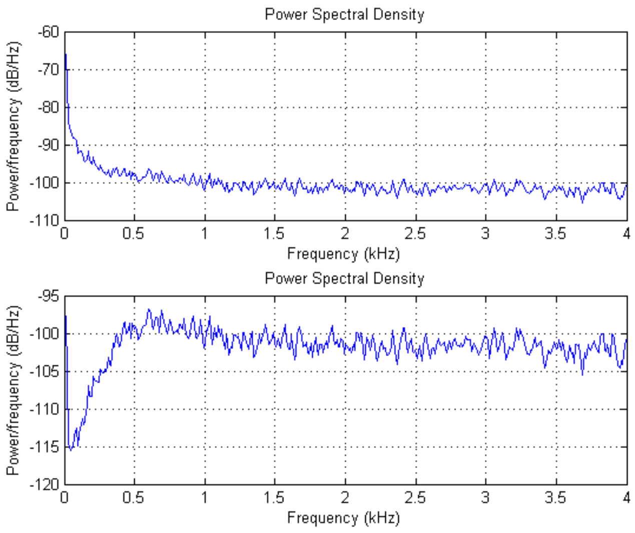


Figure 19: Filtered and unfiltered IF signal. Speed = 0 km/h

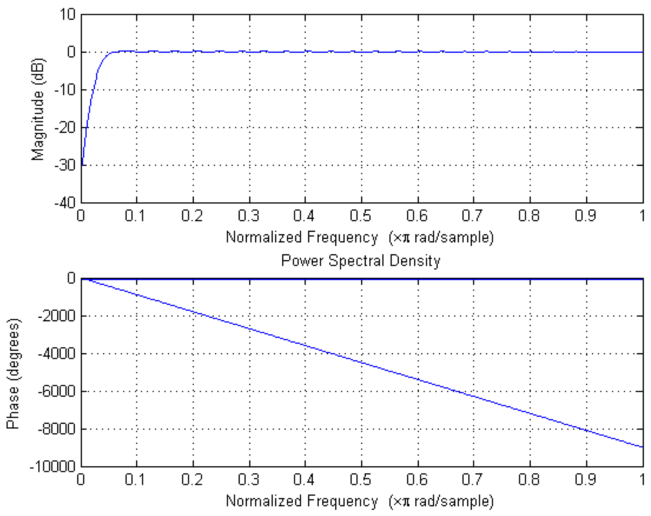


Figure 20: FIR filter frequency and phase response

5 Speed Measurement Algorithms

After the IF signal has been amplified, digitized and filtered, an algorithm to calculate the correct speed must be applied. The next sections will present different speed measuring algorithms that can be used. Section 5.4 will present some simulations done in Matlab to test these different algorithms, and Section 6.11 will describe how they were implemented on the LPC4357 microcontroller.

Intentionally, a complimentary set of algorithms, and a more accurate estimate of the Doppler spectrum was to be provided through a second Master's thesis written by a fellow student. However, those results were not ready in due time, and hence, only the algorithms described in this section were implemented and tested, using the Doppler spectrum estimate from Section 2.4.4 in the last algorithm.

5.1 Selecting the Strongest Frequency Component

The simplest way to try and estimate the vehicle speed is to simply pick the strongest component from the FFT transform, or in this case, the periodogram. This frequency should represent the correct speed, but earlier research have shown that this is not accurate enough as the FFT spectrum exhibits high variance [1]. An experiment with a 61 GHz radar was conducted, and the results can be seen in Figure 21. From this figure, it can be seen that the speed measured by the maximum of the spectrum (crosses in the figure) varies greatly compared to the true speed (curve 1).

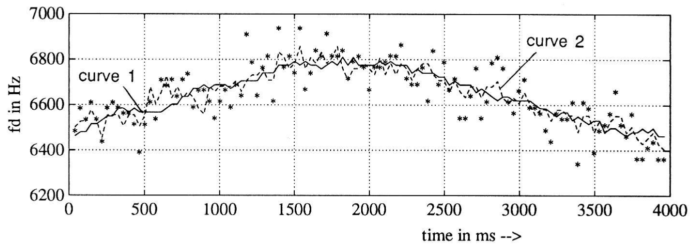


Figure 21: Estimated Doppler frequencies vs time. Speed is approximately 70 km/h [1]

Algorithm 1 shows the steps of this very simple algorithm.

Algorithm 1 Speed measurement by Strongest Periodogram Component

```
1: Estimate a single periodogram of the IF signal
2: Pick the Strongest Component
3: Calculate Speed from the Component's frequency
```

When the strongest frequency have been found, Equation (2.14) is inverted and used to calculate the corresponding speed:

$$\begin{aligned} f_d &= \frac{2v}{\lambda}\cos\alpha \\ v &= \frac{f_d\lambda}{2\cos\alpha} \end{aligned} \tag{5.1}$$

The next section will describe an enhanced version of this algorithm, where a better power spectrum estimate is calculated before selecting the strongest frequency component.

5.2 Selecting the Strongest Component from a Welch Estimate

Instead of using a single periodogram and picking the strongest component, better estimates of the Power Density Spectrum can be made using the Welch method described in Section 2.5.3. As described, this method reduces the variance of the spectrum estimate, and should therefore yield a better estimate of the vehicle speed. The rest of the algorithm is similar to the first, in the way that the strongest component of the Welch estimate is converted to the speed by using Equation (5.1). This algorithm will also be referred to as the Direct Welch method.

Algorithm 2 Speed measurement by Welch Power Spectrum Estimates

```
1: Perform a Welch Power Spectrum Estimate on the IF signal
2: Pick the Strongest Component
3: Calculate Speed from the Component's frequency
```

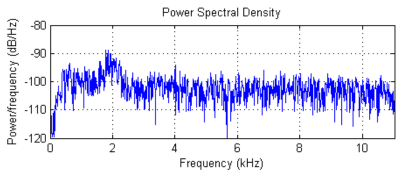
As the IF signal is not a realization of a statistically stationary process, the length N in the estimate cannot be chosen to be too long. This is because the vehicle does not travel at a constant speed or over constant ground conditions. Acceleration or deceleration can impact the speed with several m/s at every second (a car accelerating from 0 to 100 km/h over 10 seconds has an average acceleration of 2.8 m/s^2), so the time needed to sample the IF signal must be kept short to minimize distortion of the spectrum. Changing road conditions will also affect the spectrum, because of the different radar cross sections of the different surfaces.

The maximum number of points in the FFT function provided by the ARM CMSIS DSP library is 2048, so the length M is set to 2048 in all cases to get the best possible frequency resolution. Overlaps of 0%, 50,% and 75% was tested, as well as sequences of 2, 5 and 10 times the length of M. This results in a maximum total sampling time of $2048 \cdot 10/22050$ Hz $= 0.93$ seconds. Table 3 lists the different cases that was used during Matlab simulations, that will be shown in Section 5.4.

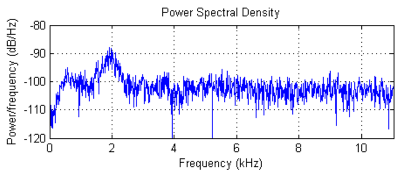
Case	K	Total Samples	Overlap	L	Sampling Time
1	2	4096	0 %	2	0.19 s
2	2	4096	50 %	4	0.19 s
3	2	4096	75 %	8	0.19 s
4	5	10240	0 %	5	0.46 s
5	5	10240	50 %	10	0.46 s
6	5	10240	75 %	20	0.46 s
7	10	20480	0 %	10	0.93 s
8	10	20480	50 %	20	0.93 s
9	10	20480	75 %	40	0.93 s

Table 3: Welch Power Density Estimate Parameters. K: Number of segments without overlap. L: Number of M-length segments with overlap.

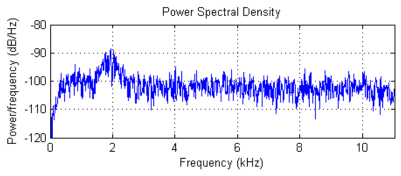
Figure 22 shows plots of the different cases of the Welch estimate from Table 3 applied to an input signal where the speed of the vehicle is approximately 70 km/h. Here, one can clearly see how the fraction of overlap and total number of samples affect the quality of the power spectrum estimate. As more samples and higher percentage of overlap are used, the variance of the estimations seem to decrease, as described in Section 2.5.3.



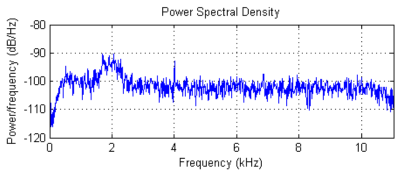
(a) Case 1



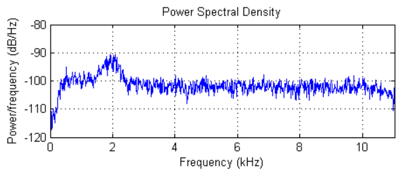
(b) Case 2



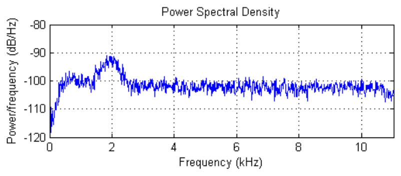
(c) Case 3



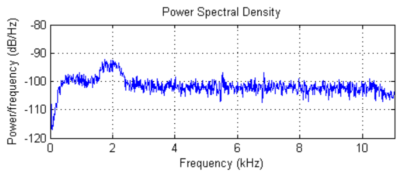
(d) Case 4



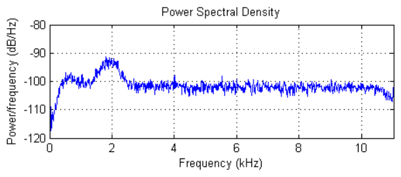
(e) Case 5



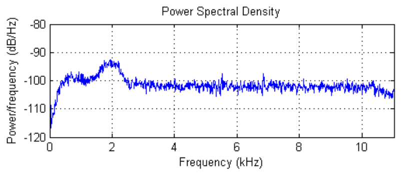
(f) Case 6



(g) Case 7



(h) Case 8



(i) Case 9

Figure 22: Plots of the Welch Power Spectrum Estimation performed on an input signal where the speed is approximately 70 km/h

5.3 Correlating with Pre-Estimated Power Spectra

The method suggested by [1] and briefly described in Section 1.4.3 uses correlation with multiple pre-estimated power spectra to find the spectrum, and hence the speed, with the best possible match. The estimate of the IF signal power spectrum is done by the Welch method as described in Section 2.5.3, but the speed is not simply calculated from the strongest component of the spectrum as in the previous algorithm. Instead, the spectrum is compared (or correlated) with estimated power spectra for different speeds.

The pre-estimated spectra are calculated using the estimation method from Section 2.4.4. A Matlab script is used to produce these estimated spectra, and save them to a file that can read by the measurement system using the microSD card interface.

As described in Section 2.5.4, a time-domain correlation is equivalent to a frequency-domain multiplication. Hence, a simple method to find the correlation between the measured and estimated spectra can be described as:

$$P_{xy}^{i}(f) = P_{xx}(f) \cdot P_{yy}^{i}(f)* \tag{5.2}$$

where P_{xx} is the Welch estimate of the IF signal spectrum, P_{yy} is the pre-estimated power spectrum, i is the index of the different pre-estimated spectra, and * denotes the complex conjugate. As both P_{xx} and P_{yy} are real-valued, the complex conjugate of P_{yy}^{*} is the same as P_{yy} itself.

When P_{xy}^{i} has been found, the values are summed to form an estimate of the total correlation, $C(i)$, of the two spectra. High values indicates that the spectra are closely correlated, and low values that they are less correlated.

$$C(i) = \sum_{f=0}^{M/2+1} P_{xy}^{i}(f) \tag{5.3}$$

When this procedure is repeated for all the pre-estimated spectra, the index of the maximum value in $C(i)$ will correspond to the pre-estimated spectrum with the best match (highest correlation). The result of this algorithm will therefore be the speed used to estimate P_{yy}^{i}, as shown in Algorithm 3.

As described in Section 1.2.2, the system should be able to measure speeds up to 250 km/h with a resolution of at least 0.5 km/h. To satisfy these specifications, the number of pre-estimated power spectra must be at least $250/0.5 = 500$.

Figure 23 shows one simulation where an input signal is compared with 500 spectra, and the input spectrum P_{xx}, $C(i)$ and the P_{yy} spectrum with the best match are plotted. Here, it can be seen that $C(i)$ produces a curve where the maximum point stands out much more clearly than the maximum of P_{xx} itself, confirming the findings presented by [1].

Algorithm 3 Speed measurement by Power Spectrum Correlation

```
1: Pre-estimate power spectra for a given number of speeds
2: Perform a Welch Power Spectrum Estimate on the IF signal
3: for $i = 0 \rightarrow$ Number of pre-estimated spectra do
4:     $P_{xy} \leftarrow P_{xx} \cdot P_{yy}^{i}$
5:     $C(i) \leftarrow \mathrm{sum}(P_{xy})$
6: end for
7: index $\leftarrow$ index of max$(C(i))$
8: Result $\leftarrow$ Speed corresponding to $P_{yy}^{index}$
```

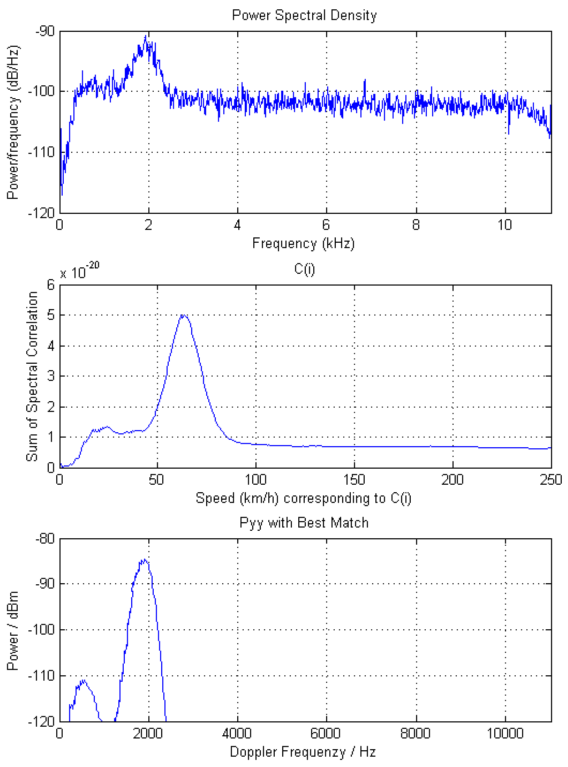


Figure 23: Simulation of the spectrum correlation algorithm performed on an input signal where the speed is approximately 70 km/h

5.4 Matlab simulations

This section will present Matlab simulations where Algorithms 2 and 3 are used to find the speed of the vehicle. Algorithm 1 was not tested, as it has been shown that this simple algorithm performs poorly [1]. The IF signal used as inputs to the simulations are the signals recorded during the initial testing of the system, which will be further described in Section 7.1. The Matlab scripts used to perform the simulations can be found in Appendix B.

The simulations were done using several test scenarios that will be described in Section 7.1. The simulations shown here use an input where the vehicle starts at 0 km/h, accelerates to 50 km/h before decelerating back to 0 km/h. The initial tests were performed without a reference measurement: These simulations were done to get an early indication of how the algorithm would perform when implemented, not as a measure of accuracy. The final tests in Section 7.3 will compare real measurements with reference measurements to check the accuracy of the algorithms.

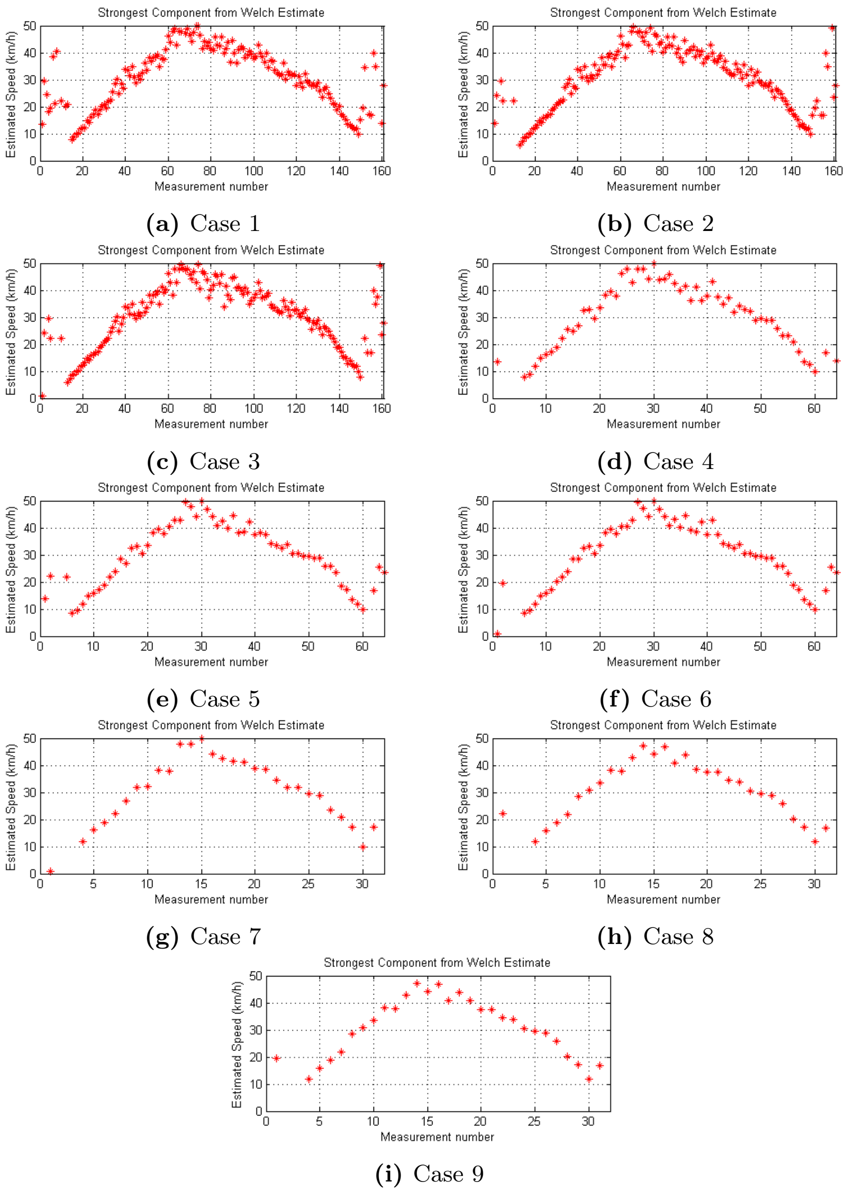
5.4.1 Speed measurement based on Algorithm 2

The different cases from Table 3 were used when simulating speed measurements using the Welch method directly. Figure 24 shows the results for the different cases. As can be seen here, the first three cases produce a lot of measurements, as the number of samples needed are only 4096, but apparently with large variance. The last three cases need 20480 samples, hence the measurements are spaced further apart. But even in Case 9, the measurements seem to inhabit a significant variance, especially around the top of the curve.

5.4.2 Speed measurement based on Algorithm 3

The measurement method based on spectral correlation was tested using the same cases as Algorithm 2. As Figure 25 shows, the variance of the results are clearly a lot smaller than when the previous algorithm was used. Even with 10240 samples used, the curve smooths out notably, and in Case 9, the results seem to be close to what one would expect. Case 9 of the Welch algorithm was thus selected as the one to be implemented in this project.

As can be seen, both algorithms struggle with measuring speeds under approximately 5 km/h, which is due to the filtering of the signal described in Section 4.3. This will be further discussed in Section 10.1.



(a) Case 1 (b) Case 2

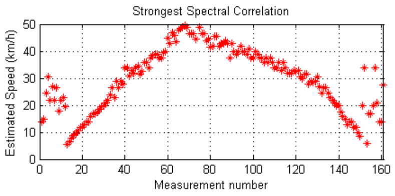
(c) Case 3 (d) Case 4

(e) Case 5 (f) Case 6

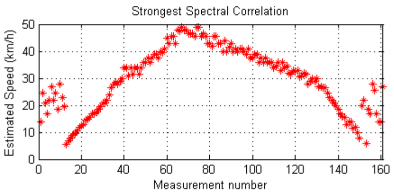
(g) Case 7 (h) Case 8

(i) Case 9

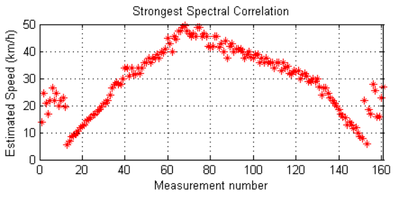
Figure 24: Speed measurement using the Welch method directly



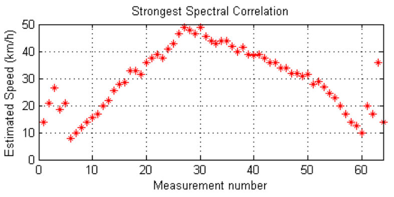
(a) Case 1



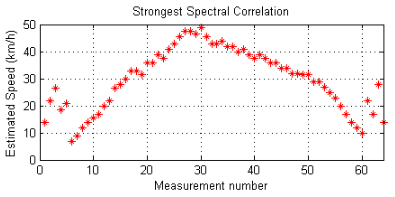
(b) Case 2



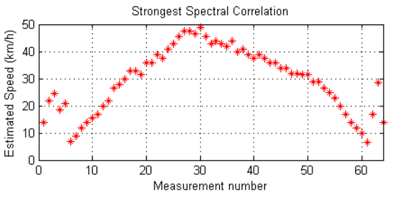
(c) Case 3



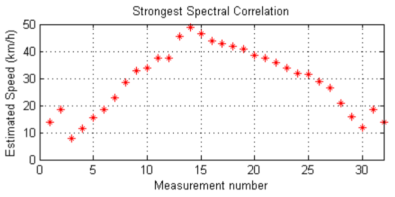
(d) Case 4



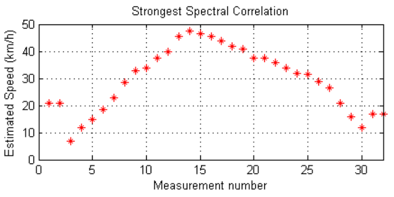
(e) Case 5



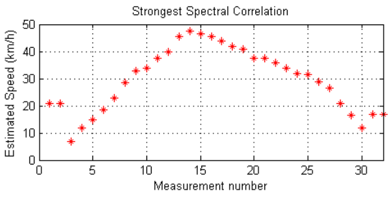
(f) Case 6



(g) Case 7



(h) Case 8



(i) Case 9

Figure 25: Speed measurement by correlating power spectra

5.5 Refresh Rate vs. Resolution

Sampling an analog signal and using the Fourier transform to convert the signal into the frequency domain always proposes the same dilemma: If a high refresh rate is desired, the time used to sample the signal must be short. But this results in fewer available samples, and this will result in lower spectral resolution. On the other hand, if high spectral resolution is desired, a high number of samples are required, which requires more time. Hence, the refresh rate becomes low. When the Welch power spectrum estimation method is used, even more samples are needed, further reducing the refresh rate. The following sections will describe a method where the samples used for the calculations allowed to overlap, using the basic FFT transform calculation as an example. The same principle can be used for the complete speed measuring calculations including the Welch estimate and spectral coherence estimations.

5.5.1 Non-overlapping Calculation

In this project, the sampling frequency used is 22,050 Hz, which is relatively low. When using a 2048 point FFT, the time needed to sample these 2048 samples will be $\frac{2048}{22050\text{Hz}} = 0.093$ seconds. When this is done, the FFT is calculated, before sampling 2048 new samples. Hence, the maximum possible refresh rate should be around 10 Hz, and this does not include the time to actually calculate the FFT and other necessary signal processing. Figure 26 shows the time line for such a non-overlapping calculation of the frequency spectrum.

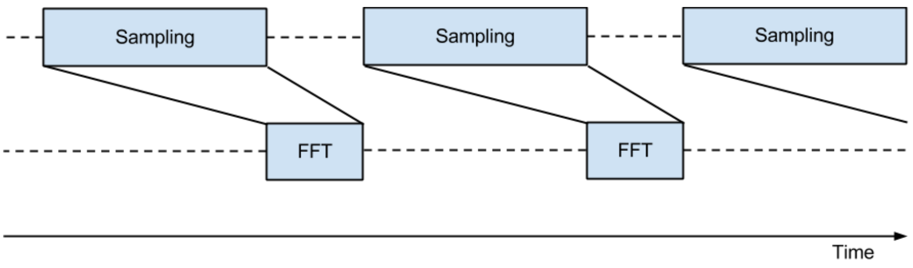


Figure 26: FFT Transform with non-overlapping samples

One might think that increasing the the sampling frequency will solve the problem, but this is not true. If the sampling frequency is doubled, the time to sample the 2048 samples will be twice as short, but these samples will now form 2048 points in a frequency domain which is twice as large. Hence, the spectral resolution is actually halved. To obtain the same resolution, the number of samples must be doubled to 4096, which will take just as much time to sample as 2048 samples with half the sampling frequency. Hence, the refresh rate will be the same.

5.5.2 Overlapping Calculation

When the time used to calculate the FFT and do all the necessary signal processing is much smaller than the time needed to sample the signal, the refresh rate can be increased by using overlapping windows of samples [26]. Instead of waiting for 2048 new samples to be collected before performing a new FFT, one can re-use some of the samples used in the last transform. This results in a "pipelined" form of calculation, as shown in Figure 27, and will allow the refresh rate to be increased dramatically, while retaining the same spectral resolution.

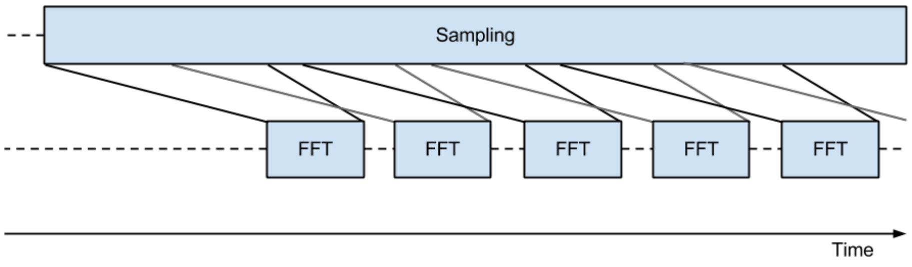


Figure 27: FFT Transform with overlapping samples

After the first 2048 samples have been recorded, FFT transforms can be performed using overlapping samples.

6 Software Development

This section will describe the software application that was developed for the speed measuring system. First, an overview of the different parts of the application will be given, next, the different libraries and drivers used to build the application will be described, and finally, a detailed description of each of the parts in the application will be given.

The application was developed in the C programming language using the ARM Keil tool chain, which includes a compiler, linker, assembler, and the μVision 4 integrated development environment (IDE). To program and debug the application, a J-Link JTAG adapter was used.

The relevant sections of application source code can be found in Appendix A. In the following sections, direct references to the source code and will be kept to a minimum. This is done to increase the readability of the thesis, and to focus on how the application is implemented. Important functions will be described in English rather than with source code examples. However, if the reader wishes to reference the code while reading these sections, it is well documented and it should therefore be easy to find and understand the correct functions / modules in the appendix. The source code for the different libraries used in the application is left out of the appendix, as this is not a product of this project. These libraries are open-source and can be downloaded from the respective developers' web sites ([3, 23]).

6.1 Application Overview and Menu System

The application that was developed consists of three main parts (modes): A spectrum analyzer developed to visually inspect the radar IF signal during testing, a speed-measuring module for signal processing and presentation of results, and a module for recording and storing the IF signal on a microSD memory-card. The application lets the user choose between these different modes via a menu system presented on the LCD display (see Figure 28). The menu system is a simple text-based system, using the input from the on-board joystick for navigation.

The application was implemented as a state machine, where the system is in different states depending on which mode it is in. In the following sections, details on the different states and modes will be given.

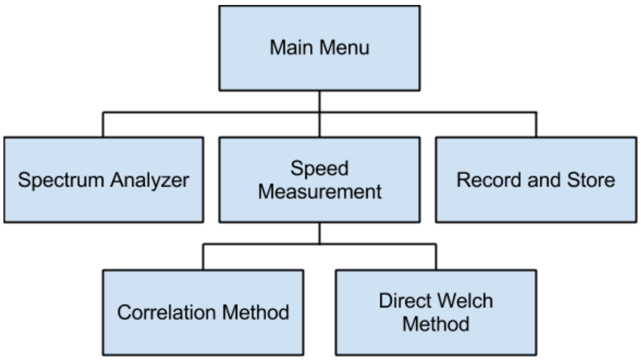


Figure 28: Application Overview

6.2 LPCOpen and ARM Libraries

To support the implementation of the application, the LPCOpen library was used. LPCOpen is an open-source platform built for the NXP LPC-series microcontrollers. It contains chip drivers, development board support drivers, example code etc [27]. The platform is constantly being developed and improved, and is verified by NXP Semiconductors. Figure 29 shows how the library is built up in different layers.

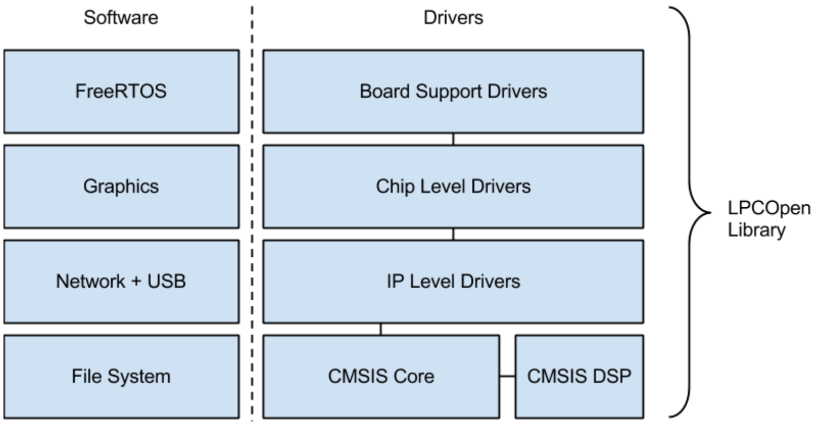


Figure 29: LPCOpen Library Structure

The ARM Cortex Microcontroller Software Interface Standard (CMSIS) is a standardized hardware abstraction layer (HAL) that provides an interface to the Cortex M processors and its peripheral units [3]. In addition to this core functionality, it provides a digital signal processing (DSP) library with over 60 fixed- and floating-point functions, e.g. functions for real and complex FFT-transforms, statistics, filtering etc. It also specifies some standard intrinsic functions to directly manipulate the processor registers, as well as standardized initialization methods. It is up to the hardware vendors, such as NXP, to implement the functions in the CMSIS interface.

The LPCOpen library is built on the CMSIS core interface, but it also contains higher-level functions and drivers. An IP (intellectual property) layer implements processor and peripheral drivers that are chip-independent, a Chip driver layer implements chip-specific functions and can use the IP drivers, and finally a board support layer implements board-specific functions, such as drivers for initializing and using displays, memory, I/O modules etc. In addition to these drivers, the library contains a real-time operating system (FreeRTOS), network and USB protocol stack, FAT file system, and the emWin and SWIM graphics libraries. It also includes the CMSIS DSP library.

When the application was implemented, some of the examples from the LPCOpen library were used as templates, but major rewriting and restructuring was necessary. For efficiency, the CMSIS DSP functions were used when applicable.

6.3 ARM CMSIS DSP Library functions

Below are brief descriptions of the most important DSP functions from the ARM CMSIS DSP library that were used in the software part of this project. For detailed description, refer to the CMSIS documentation ([3]).

`arm_rfft_f32`
Implements the Real Fast Fourier Transform for 32-bit floating-point data. Uses the Radix-4 type FFT transform to calculate the results from a real-valued input, as outlined in Section 2.5.1. The output is complex. Supported FFT lengths are 128, 512 and 2048 samples.

`arm_cmplx_mag_f32`
Computes the magnitude of the elements of a complex data vector.

`arm_fir_f32`
Implements a Finite Impulse Response (FIR) filter for 32-bit floating-point data. The FIR filter algorithm is based upon a sequence of multiply-accumulate operations, as seen in Figure 30.

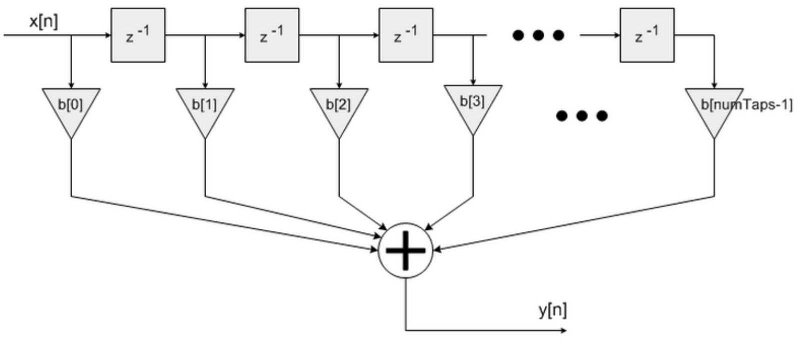


Figure 30: FIR Filter Configuration [3]

`arm_mult_f32`
Element-by-element multiplication of two vectors.

`arm_max_f32`
Computes the maximum value of an array of data. The function returns both the maximum value and its position within the array.

6.4 Initial Setup of the Development Board

Before the main application could be implemented, the MCB4357 development board had to be set up and the main components such as external memory and LCD display had to be initialized to ensure correct behavior. Fortunately, the LPCOpen library includes startup- and initialization functions for this purpose that only need to be included in and called from the main application. The `"board.h"` file from the LPCOpen library is included, and a `BoardInit()` function is used to set up the board. This function initializes all the necessary on-board hardware, and leaves it ready to be used by the application.

6.5 LCD Display Driver (`display_mcb4300.c`)

To implement a driver for the LCD display, an example from the LPCOpen library was used as a base. This included, among others, simple functions for writing characters to the display, but it was very limited and needed added functionality.

The driver uses the board- and chip-level functions from the LPCOpen library to interface with the LCD display. A frame buffer located in the external

SRAM memory contains information about all the pixels on the display, and the microcontroller's LCD controller peripheral is set up to automatically update the display with the contents of this buffer. Hence, the display driver only needs to update the frame buffer to display information on the screen.

The following list describes the most important functions of this driver:

- `display_init`: Calls the board- and chip-specific initialization functions, sets up the frame buffer, turns on LCD backlight and sets the font type and colors to use on the display.
- `display_print_string`: Prints a string of characters at the desired row and column. If the newline character `'\n'` or the end of a row is encountered, the string is continued on the next line. If the form feed character `'\f'` is encountered, the screen is cleared.
- `display_draw_line`: Draws a horizontal or vertical line of any length from any point on the display. Used to draw simple screen layouts and for printing the power spectrum from the spectrum analyzer (see Section 6.9).

In addition, the standard output used by `printf` is redirected to the display for debugging purposes.

6.6 UDA1380 CODEC driver (`uda1380_mcb4300.c`)

As shown in Section 4.1, the UDA1380 audio CODEC chip is well suited for sampling and digitizing the IF signal from the radar. Hence, it was necessary to develop a driver that could handle all the communication with the CODEC. The I^{2}C bus is used for interfacing with the CODEC's control registers and this connection has to be set up as a part of the initialization routine for the CODEC. After this connection is set up, the CODEC is programmed with the desired parameters. Table 4 shows the different settings used in the actual application. To minimize noise, all features that are not necessary are disabled if possible.

6.6.1 Sampling Frequency and Other I^{2}S Parameters

Section 3.1.1 shows that the sampling frequency should be $\sim$ 22,600 Hz. This frequency is very close to the standard sampling frequency of 22,050 Hz (half of 44,100 Hz which is used in Audio CD's, MP3 files, etc.). Hence, this frequency is chosen as the sampling frequency. This ensures flexibility of the system: If the CODEC is replaced, it will almost certainly be able to use this frequency. It is also very unlikely that speeds as high as 250 km/h will ever occur, so the loss of the top 450 Hz will not be a problem.

Feature	Setting
ADC	Left channel enabled
DAC	Disabled
LNA (Microphone input)	Enabled, with 30 dB VGA Gain
PGA (Line input)	Disabled
DC Filter	Enabled
All Volume Controls	Set to 0 dB
Headphone Driver	Disabled

Table 4: UDA1380 CODEC settings

The I^{2}S peripheral unit can handle 8, 16, or 32 bit data. As the AD converter in the CODEC is 24 bit, 32 bit is selected as the I^{2}S data word length, and as only the left channel is used, the peripheral is set to work in mono mode (only the left channel data is placed in the receive buffer).

6.6.2 Modes of Operation

After the CODEC has been programmed, the microcontroller's I^{2}S peripheral is set up to be able to receive the data from the CODEC. An interrupt routine is used to receive the data: When the buffer for incoming data is half full (4 32-bit words), the interrupt routine is invoked and the data is stored. Depending on the mode of operation, two different methods for storing the data are used:

- `UDA1380REC`: When in record mode, the driver is set up to store the data in a single sequential memory chunk. The external memory is used for this purpose, and it will allow the driver to record data until the system is out of memory. This mode is used when storing data on the microSD card (see Section 6.10).
- `UDA1380STREAM`: When in streaming mode, the data is stored in a ring-buffer. This ensures that the system will never run out of memory, but the oldest data will always be replaced by the newest, meaning that data will be lost if it is not copied to another location before it is overwritten. This mode is used by the spectrum analyzer and the speed-measuring algorithms, where snapshots of the ring-buffer are used in the calculations (see Sections 6.9 and 6.11).

6.6.3 Important Functions

These are the most important functions of the CODEC driver:

- `uda1380_init`: Sets up the I^2C and I^2S connections. Programs the CODEC with the correct settings.
- `uda1380_start_rec`: Starts to record a specified number of samples to a specified memory location.
- `uda1380_stop_rec`: Stops the recording.
- `uda1380_start_stream`: Starts to stream from the CODEC to the ring-buffer.
- `uda1380_stop_stream`: Stops the streaming.

6.7 SD/MMC driver (`sdmmc_mcb4300.c`)

The SD/MMC (Secure Digital / Multimedia Card) interface included on the NXP LPC4357 chip supports reading from and writing to memory cards. The MCB4300 board contains a microSD card slot, which is used to store recorded data.

6.7.1 File System

To utilize a memory card, a file system is necessary. The LPCOpen library contains an example where the FAT file system is used. This file system is one of the most widely used file systems, and can be read from and written to by almost any computer regardless of the operating system. The file system is included with the LCPOpen library and was set up following an example, but some functions had to be custom built to fit this application.

6.7.2 Wave File Format

To be able to load the recorded IF signal into Matlab for simulation purposes, or even listen to the signal with an audio player, a file format had to be chosen. As the data that is received from the CODEC is on 32 bit two's complement format, this can be directly used as Pulse-code modulated (PCM) audio data. The PCM format is a standard audio format used in computers, CD's, etc. To store PCM data, a Waveform Audio File (.WAV) can be used. Wave files follow the Resource Interchange File Format (RIFF) container specification, which specifies a file header with different blocks (chunks) containing information about the file. An audio player or another program can read this header to interpret the data format, sampling frequency, etc.

For the 32-bit PCM data in this application, the RIFF header is shown in Table 5. The header is written before the data in the file, with the strings "RIFF", "WAVE", "fmt " and "data" written as ASCII characters, and other values written as 16 or 32 bit numbers.

Header field	Value
RIFF Chunk	`"RIFF"`
Chunk size	Header size + Data size (bytes)
WAVE ID	`"WAVE"`
FORMAT Chunk	`"fmt "`
Chunk Size	16
WAVE Format	1 (PCM)
Channels	1 (mono)
Samples per Second	22050
Bytes per Second	22050 * 4
Bytes per Sample	4
Bits per Sample	32
DATA Chunk	`"data"`
Chunk Size	Data Size (bytes)
Data	PCM Data

Table 5: Wave File Header for 32-bit mono PCM data

6.7.3 Important functions

The following are the most important functions from the SD/MMC driver:

- `sdmmc_init`: Initializes the SD/MMC interface, sets up and mounts the file system. It also starts the Real-Time Counter (RTC) which is used for time stamping by the FAT file system.
- `sdmmc_save_wav`: Saves a number of bytes of 32-bit PCM data to a Waveform Audio File (.wav). Sets up and writes the RIFF header before writing the data.
- `sdmmc_load_float`: Loads a file containing 32-bit floating-point values to a specified location in memory.
- `sdmmc_save_speed`: Saves speed measurement values to file, including a header with measurement type and time-stamp at the beginning of the file.

6.8 DSP Functions (`dsp_funcs.c`)

Both the spectrum analyzer and the speed-measurement functionality need to perform some sort of digital signal signal processing on the radar signal, as they both depend on knowing the frequency components of the signal. As described in Section 6.3, the ARM CMSIS interface contains a number of DSP functions. However, additional processing is needed, and hence some custom DSP functions were developed and collected in the file `dsp_funcs.c`. This section will describe the purpose of these functions and how they were implemented.

`dsp_periodogram`
This function calculates the power density spectrum (periodogram) of the IF signal received from the radar module, as described in Section 2.5.2, and returns a pointer to the buffer with the output values. It uses the CODEC driver to sample the signal, and the ARM CMSIS DSP library functions to do the main signal processing calculations. As described in Section 5.5, doing calculations using overlapping sample segments ensures a high refresh rate while keeping a high resolution, and this principle can be exploited by this function as it uses snapshots of the samples in the ring-buffer to do the calculations. The overlap is determined by how often the function is called.

The function assumes that the CODEC has been started in the streaming mode (see Section 6.6) before it is called. This means that the ring-buffer will contain a set of the newest samples of the signal. Before the calculations are started, a snapshot of the ring-buffer is taken by copying the contents to another memory location. During this process, interrupts are disabled to ensure that the buffer is not overwritten while it is being copied.

After the samples have been copied from the ring-buffer, the function implements Equation (2.28) to calculate the power density spectrum using, the `arm_rfft_f32` function with 2048 points to calculate the FFT and the `arm_cmplx_mag_f32` function to calculate the magnitude of the complex result of the FFT.

As the spectrum is symmetric around 0 and mirrored around $F_s/2$, all components except 0 and $F_s/2$ are doubled, and only the one-sided spectra from 0 to $F_s/2$ is returned, resulting in 1025 frequency components.

`dsp_welch`
The function for calculating the Welch power spectrum estimate is similar to the `dsp_periodogram` function in the way that it needs the CODEC to run in streaming mode and how it copies samples from the ring-buffer. The total

length N of the Welch estimate determines how many sampled that are copied from the buffer.

When the samples have been copied from the buffer, the signal is filtered by the `arm_fir_f32` using the filter described in Section 4.3. For each overlapping segment, $M = 2048$ samples are windowed by multiplying with the Hamming window.

The Hamming window function can be described as

$$w(n) = 0.54 - 0.46 \cos \frac{2\pi n}{M-1} \qquad [2] \tag{6.1}$$

thus, the scaling factor U calculated from Equation (2.33) becomes

$$U = \frac{1}{2048} \sum_{n=0}^{2047} w^2(n) = 0.3972 \tag{6.2}$$

After the segments have been windowed, the periodogram is calculated by using the `arm_rfft_f32` and `arm_cmplx_mag_f32` functions. The scaling factors $1/MU$ and $1/L$ from Equations (2.32) and (2.34) are applied, and the modified periodograms are summed as shown in Equation (2.34).

Finally, a pointer to the buffer with the result are returned. Similarly to the `dsp_periodogram` function, the components between 0 and $F_s/2$ are doubled, and only the one-sided spectra from 0 to $F_s/2$ is returned.

6.9 A Simple Spectrum Analyzer (`spectrum.c`)

The spectrum analyzer utilizes the display and CODEC drivers, as well as the `dsp_periodogram` function from `dsp_funcs.c`. It is started by calling an initialization function which sets up the display and starts the CODEC in streaming mode. Once initialized, the `spectrum` function can be called whenever the spectrum should be updated on the display. By varying the period of which this function is called, the update frequency on the display varies accordingly. An update frequency of at least 20 Hz is recommended if a "real-time" appearance is desired. Figure 31 shows a photo of how the spectrum analyzer appears on the screen of the MCB4300 development board.

6.9.1 Drawing the Spectrum

After the power spectrum has been calculated by the `dsp_periodogram` function, the spectrum is displayed on the screen. As the screen is only 320 pixels wide, every single one of the 1025 spectrum components cannot be

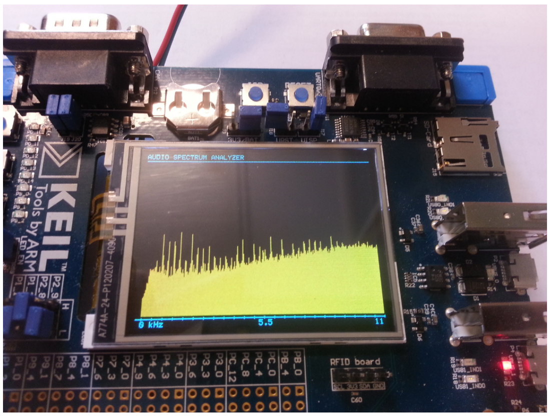


Figure 31: Photo of spectrum analyzer display

displayed at once. To solve this, adjacent components are averaged and merged into one frequency bin so that the spectrum to be displayed can fit on the display. When this is done, each bin is drawn as a vertical line on the display using the `display_draw_line` function from the display driver. A simple state diagram for the spectrum analyzer is shown in Figure 32.

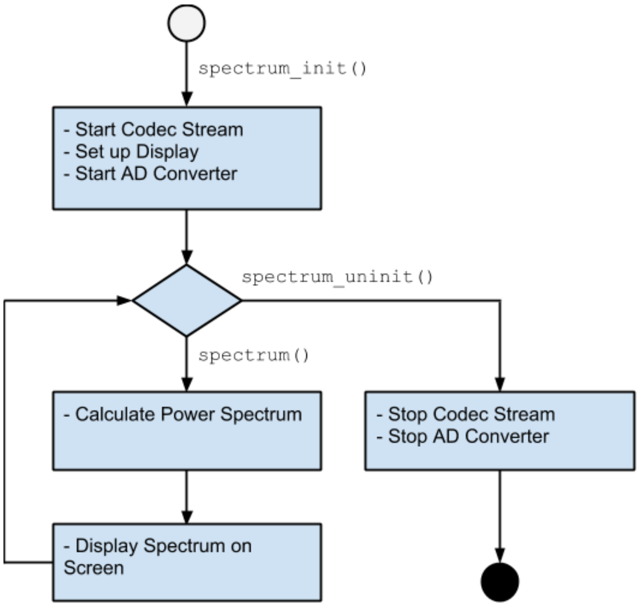


Figure 32: Spectrum Analyzer State Diagram

The spectrum is displayed with a dB scale on the vertical axis, with 0 dB as the maximum value and -80 dB as the minimum value. The 0 dB reference can be manually adjusted, as will be described in the next paragraph.

6.9.2 Adjusting the Sensitivity

To adjust the sensitivity, or more specifically the 0 dB reference used in the spectrum analyzer, a potentiometer on the MCB4300 board can be used. This is connected to one of the analog inputs of the microcontroller, and the value from the AD converter is used to set the sensitivity. This way, when signal is weak, the spectrum analyzer can be "zoomed" in to better display the various frequencies.

6.9.3 Important functions

The following are the most important (and the only public) functions from the `spectrum.c` file:

- `spectrum_init`: Sets up the display for the spectrum analyzer. Starts the UDA1380 CODEC in streaming mode. Starts the AD converter that is used for setting the 0 dB reference.
- `spectrum`: Calls the `dsp_periodogram` function to calculate the power spectrum. Displays the spectrum on the display by utilizing internal functions to merge frequency bins and calculate the dB value before using display driver functions to draw the spectrum.
- `spectrum_uninit`: Stops the UDA1380 CODEC and the AD converter.

6.10 Recording and Storing the raw IF Signal

One of the functions available from the application's main menu is to record and store a portion of the received IF signal. By storing the data on a microSD card, it is possible to use the raw unprocessed samples in simulations and tests on a computer.

When the "Record and Store" function is selected, the UDA1380 driver is used to start the CODEC in record mode. The recording can be stopped by pushing a button on the development board. If the recording is not stopped manually, the application will record until the memory destination is full. Figure 33 shows a state diagram of how the recording process behaves.

When the recording is finished, the user is prompted to enter a name for the file. The name can contain any characters from 'a' to 'z' and numbers from 0

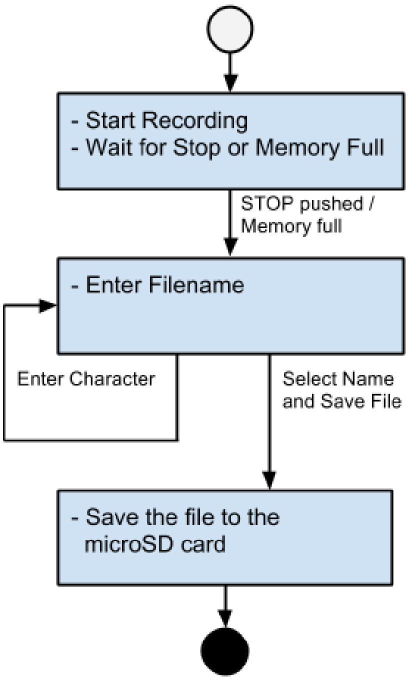

- Start Recording
- Wait for Stop or Memory Full
STOP pushed / Memory full
- Enter Filename
Enter Character
Select Name and Save File
- Save the file to the microSD card

Figure 33: "Record and Store" State Diagram

to 9. As the development board does not include a keyboard, the characters can be entered by scrolling through them with the on-board joystick. When the desired character is found, the joystick is moved to the next position, or pushed to save the file with the entered name.

To save the file, the `sdmmc_save_wav` function from the SD/MMC driver is used. The function takes the name of the file and the location of the data as parameters, and saves the file to the memory card.

6.11 Implementation of Speed Measurement Algorithms (`speed.c`)

The actual speed calculation was implemented as a single function, using a `switch` case to select the appropriate calculations based on the type of algorithm that is selected. The two different settings are `SPEED_WELCH` and `SPEED_CORRELATION`, corresponding to Algorithms (2) and (3). The type is set during the initialization phase of the speed measurement module. The next paragraphs will describe the different functions in this module in greater detail.

speed_init
When the initializing function is called, it sets up the display to present the speed measurements. It then starts the CODEC in streaming mode, and based on the type of algorithm selected, it initializes the necessary variables and buffers. When `SPEED_CORRELATION` is selected, the pre-estimated P_{yy} spectra are loaded from the microSD card to the memory using the `sdmmc_load_float` function. If the file is not found, an error message is displayed, and the measurement is aborted.

In addition to this, the function also takes a time-stamp from the real-time clock to mark the start of the measurements, and sets up the file name to use when storing the data (see Section 6.12). Figure 34 shows a block diagram that outlines the behavior of the initialization function.

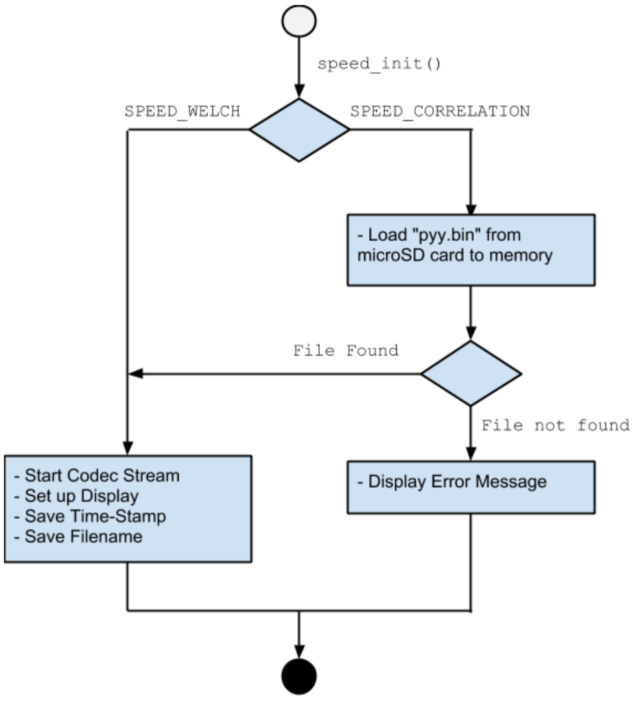


Figure 34: Speed Module Initialization Block Diagram

speed_deinit
The de-initializing function stops the CODEC, takes a time-stamp to mark the end of the measurement and saves the logged speed values to a file on the memory card using the `sdmmc_save_speed` function (see Section 6.12).

speed
This is the main function of the speed measurement module. It implements Algorithms 2 and 3 described in Section 5. The `dsp_welch` function is used to perform the Welch power spectrum estimate, with $N = 20480, M = 2048$ and a 75 % overlap. The ARM CMSIS DSP functions `arm_mult_f32` and `arm_max_f32` are used to do the correlation and find the maxima of P_{xx} or $C(i)$, respectively. Figure 35 shows a block diagram of the behavior of the `speed` function.

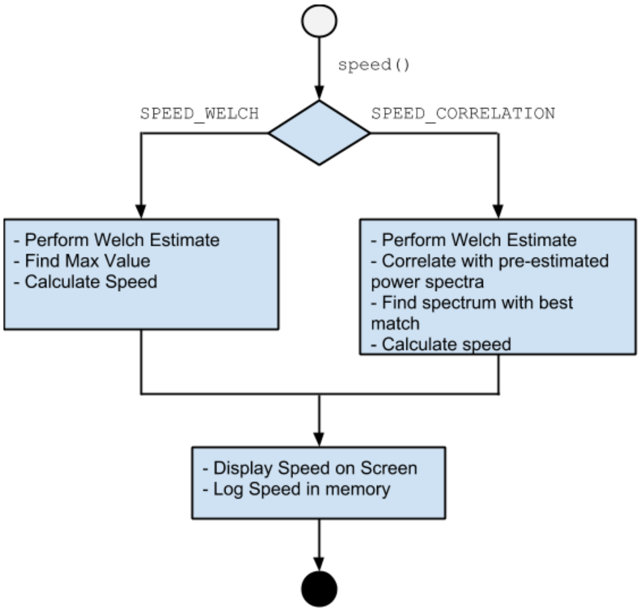


Figure 35: Speed Measurement Block Diagram

As described earlier, the DSP functions used here are able to perform calculations using overlapping samples. Hence, the `speed` function can be called as often as needed/possible, to allow for as many measurements per second as possible.

After a speed has been calculated, it is presented on the screen as seen in Figure 36.

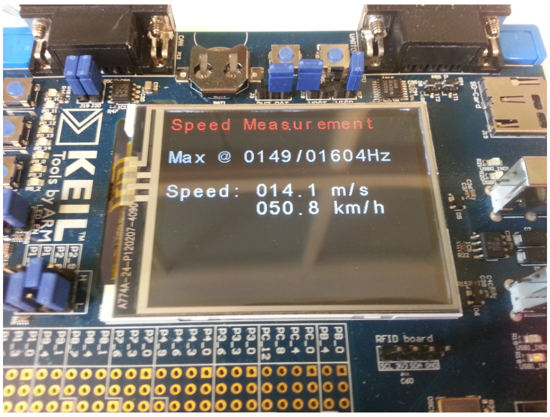


Figure 36: Photo of speed measurement display

6.12 Logging and Storing the Speed Measurements

When a speed measurement has been performed, the result has to be stored. This was solved in the following manner:

1. When the measurement is started, a time-stamp from the real-time clock is saved to mark the starting time of the measurement.
2. During the measurement sequence, each result is saved to a buffer in the external memory
3. When the measurement is stopped, a second time-stamp is saved to mark the end of the measurement.
4. Finally, the results are composed into a single file and saved to the microSD memory card.

As the interval between single measurements is constant, the start and end time-stamps can be used to find the point in time for each single measurement, by dividing the total time with the total number of measurements to get the time-distance between them.

6.12.1 File Format

The file format used to store the recorded speed measurements is presented in Table 6. The first 16 bytes are the header of the file and contains the type of measurement (`SPEED_WELCH` or `SPEED_CORRELATION`) as well as the start and end times. The file extension is ".DOP".

Byte Number	Content	Format
1	Measurement Type	8-bit Unsigned Integer
2	Start Time Hour	8-bit Unsigned Integer
3	Start Time Minute	8-bit Unsigned Integer
4	Start Time Second	8-bit Unsigned Integer
5, 6	Start Time Year	16-bit Unsigned Integer
7	Start Time Month	8-bit Unsigned Integer
8	Start Time Day	8-bit Unsigned Integer
9	End Time Hour	8-bit Unsigned Integer
10	End Time Minute	8-bit Unsigned Integer
11	End Time Second	8-bit Unsigned Integer
12, 13	End Time Year	16-bit Unsigned Integer
14	End Time Month	8-bit Unsigned Integer
15	End Time Day	8-bit Unsigned Integer
16 → EOF	Speed Measurements	32-bit Floating Point

Table 6: File format for logged speed measurements

A Matlab script was developed to read and plot the speed measurements on a computer. This script can be found in Appendix B.

6.13 Memory Management

In any computer program, memory management is critical, and this application is not an exception. When using external memory in addition to the on-chip memory, the memory address space has to be managed manually. Hence, a memory map was developed to ensure that there were no overlapping memory segments or illegal addresses used.

As mentioned in Section 3.2.1, the microcontroller itself only contains 72 kB of local SRAM memory, with up to 136 kB of total on-chip memory. As seen from Table 7, the frame-buffer alone exceeds this limit, and when adding the other large buffers, it is clear that extra external memory was necessary. The table also shows the most memory-using buffers, and the total memory requirement for these buffers. Clearly, a lot of other variables and small buffers increases the system-wide memory requirement.

The MCB4357 development board contains 16 MB of external SDRAM. Table 7 also shows the memory map that was used in the application. The NXP LPC4357 data sheet [22] was used to find the correct address range to use. In short, the external memory address space starts at `0x2800 0000` and ends at `0x28FF FFFF`.

Address	Size	Comment
`0x28000000`		Start of external memory
`0x28000000`	153600 B	Frame-buffer for display
`0x28027800`	8192*2 B	Buffer for FFT output (Complex)
`0x2802B800`	8192 B	Magnitude buffer
`0x2802D800`	8192 B	Window function buffer
`0x2802F800`	40960 B	Welch input signal
`0x28057800`	40960 B	Filtered input signal
`0x2807F800`	40960 + 8 B	Ring buffer
`0x280A7808`	4100 B	PXX buffer
`0x280A880C`	4100 B	PXY buffer
`0x280A9814`	2 050 000 B	PYY buffer $(1025 \cdot 500 \cdot 4)$
	522 456 B	Subtotal
`0x28300000`	12 MB	Space left for recording
`0x28FFFFFF`		End of external memory

Table 7: Memory Map

The last 12 MB of the buffer is reserved for the CODEC's recording buffer. This enables the application to record sequences of raw data samples up to almost 2.4 minutes:

$$\frac{12\text{MB}}{(22050 \cdot 4)\text{B/s}} = 142\text{s} \tag{6.3}$$

7 Testing the System

To test the system, the equipment (development board, power supply and radar) was set up and mounted on a test vehicle. Different positions for the radar was tested, as shown in Figure 37.

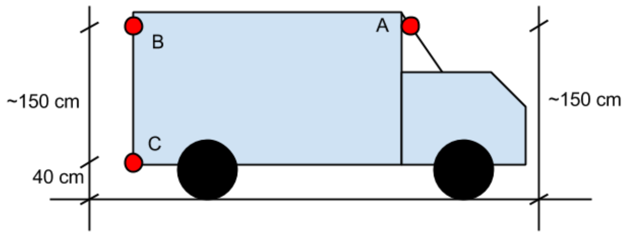
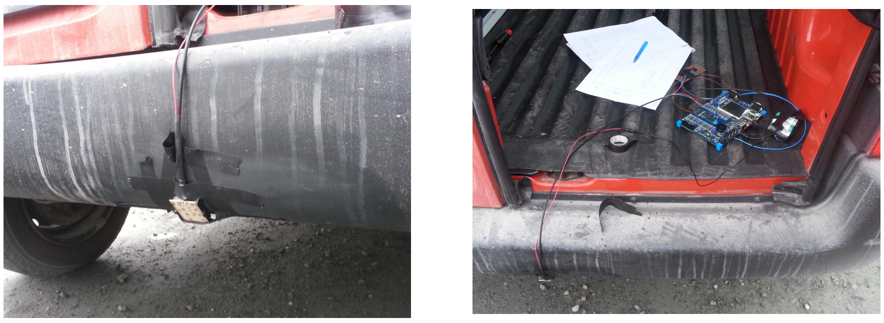


Figure 37: Possible radar sensor positions on vehicle

The vehicle was provided by Q-Free ASA, and contained equipment for accurate reference measurements using a GPS system. Figure 38 shows some pictures of the radar setup on the actual test vehicle.



(a) Close-up photo of the radar sensor mounted on the rear bumper

(b) Photo of the different parts of the system at the test vehicle

Figure 38: Photographs of the radar test setup

The testing of the system was done in two stages: The first stage was to record raw data used in the Matlab simulations to test the different speed measuring algorithms as described in Section 5.4. The second stage was to do real speed measurements using the developed algorithms. The next sections will describe these two stages in more detail.

A 45° radar tilt, θ, was used in all the tests. This angle was selected based on the findings in [8], which suggests angles within 30 to 45 degrees, and the back-scattering coefficients of asphalt shown in Figure 6.

7.1 Initial Tests (Data Recording)

The first tests were done after the development of the recording functionality of the system (described in Section 6.10) had finished. The system was set up as described in the previous section, and a number of tests were executed. The same test cases were used for both the high positions (A and B), and the low position (C). Table 8 shows the different test scenarios. All scenarios were tested for all the mounting positions. The tests were performed on dry asphalt roads.

Test nr	Scenario
1	20 km/h constant speed.
2	40 km/h constant speed
3	60 km/h constant speed
4	70 km/h constant speed
5	Acceleration/deceleration to and from 50 km/h

Table 8: Initial test scenarios

No speed measuring algorithms were used for these tests, as the goal was to record raw signal data for use in Matlab simulations. The IF signal was simply sampled and stored directly to .wav format files on the microSD card, as described in Section 6.10.

7.2 Initial Test Results

After the initial tests had been completed, the results were imported into Matlab to be used in speed-algorithm simulations. It became evident from power spectrum estimations that the high positions (A and B) does not provide a sufficiently strong signal to be used for speed measurements. Figure 39 shows a comparisons of the spectra from the high and low positions, where the vehicle is moving at approximately 40 km/h. Hence, only the low position, C, was used during the final tests that will be described in the next section.

7.3 Final Test of Speed Measuring Algorithms

The final tests of the speed-measuring system was done with the same radar setup as the initial tests, but only position C was used. Both the direct Welch

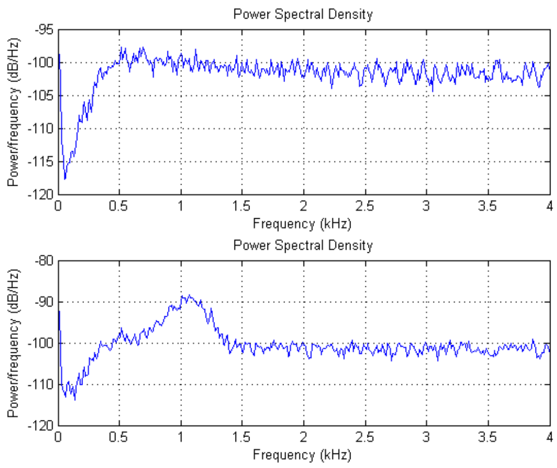


Figure 39: Doppler spectra for high (upper) and low mounting positions. Speed = 40 km/h.

method (Algorithm 2) and the spectral correlation method (Algorithm 3) were tested.

The system was tested on a route in and around the city of Trondheim, Norway, with speeds varying from 0 to approximately 80 km/h. The road conditions were mostly dry, but some moisture resided from a rainfall earlier in the day. All the roads were covered with asphalt. A GPS reference system was used to measure the true speed of the vehicle during the whole duration of the test. Unfortunately, Q-Free ASA was not able to provide data on the accuracy of this system, but it will be assumed that it is at least as accurate as the GPS system described in Section 1.3.2.

The tests were done in the time period between 03-Jun-2013 12:33:14 (UTC) and 03-Jun-2013 14:47:14 (UTC). Four different test runs were completed, the first two using Algorithm 3, and the latter two using Algorithm 2, to measure the speed of the vehicle. The different test runs are shown in Table 9. Figure 40 show the complete test route. The results of the final tests will be presented in Section 9.

Run nr	Alg.	Duration	Scenario
1	3	949 s	Route from Ranheim to Trondheim city center, using urban roads with many intersections and relatively low speeds.
2	3	1479 s	Route from Trondheim city center to Nardo, including periods with traffic jams and highways with relatively high speeds.
3	2	658 s	Route within Trondheim city center, using urban roads with many intersections and relatively low speeds.
4	2	1008 s	Route from Nardo to Ranheim, including periods with traffic jams and highways with relatively high speeds.

Table 9: Test run description for speed measurement tests

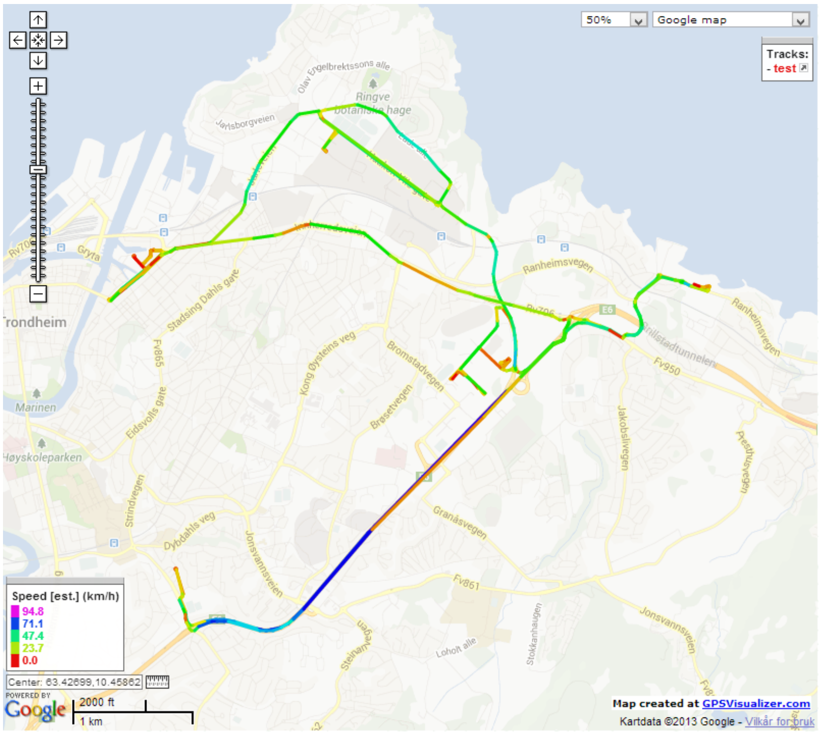


Figure 40: Map of the route driven during speed measurement tests

8 Statistical Analysis and Error Correction

This section will describe the method that was used to align the measurement values with the reference data, as well as the statistical properties used to describe the accuracy of the results.

8.1 Aligning Measurements with Reference values

To be able to compare the measured speeds with the reference speed from the reference GPS system, the samples need to be aligned. The reference system uses a 1 Hz sampling rate, so the speed measurement samples need to be down-sampled to the same frequency.

This down-sampling was done in Matlab using interpolation and decimation. First the sampling rate of the measurements was calculated by measuring the period between the samples. For the samples taken using the correlation algorithm, this frequency was 2.922 Hz. The measurements were down-sampled using the following method:

1. Interpolate (up-sample) by a factor of 1000
2. Decimate (down-sample) by a factor of 2922

This method gives a total down-sampling factor of 2.292, ensuring that the samples will match the 1 Hz sampling frequency of the reference measurements. The same method were used to down-sample the measurements where the other algorithm was used.

8.2 Analyzing The Results

After the samples from the reference system and the radar measurements had been aligned, they could be compared to analyze the error of the measurements.

To describe the accuracy of the results, the following quantities are used: Mean error in km/h ($\widehat{E}_{km/h}$), standard deviation in km/h ($\sigma_{km/h}$), mean error in percentage ($\widehat{E}_{\%}$) and standard deviation in percentage points ($\sigma_{\%}$). The mean errors represents the expected error of the measurement, and the standard deviation reflects the accuracy of the measurements. The errors are assumed to be normally distributed, and hence, 68.3 % of the errors falls within $\pm 1\sigma$ from the mean value, 95.4 % falls withing $\pm 2\sigma$, and 99.7 % falls within $\pm 3\sigma$.

8.3 Error Correction

As the results in Section 9.2 will show, there is a negative mean error both in km/h and %. This error is likely to stem from the fact that the radar tilt angle, θ, was not accurately adjusted when the radar was mounted on the vehicle. This will yield a measurement error proportional to the error in the tilt angle, as can be seen from Equation (5.1), meaning that the speed algorithms will match the measured Doppler spectra to the wrong speed.

To correct for this, the radar measurements were multiplied with a scaling factor to compensate for the angle error. After several trial-and-error attempts, it was found that a scaling factor of 1.18 was optimal for the results from the test runs. However, as will be shown, the error is not completely linear, as the mean error increases when measurements of lower speeds are left out of the analysis.

The results obtained after this error correction are presented in Section 9.3 and further discussed in Section 10.4.

9 Results

This section will present the results of the main system test. The results that are shown in the first sections are excerpts of the complete range measurement. This is done to present some different cases such as high speeds, low speeds, start/stop conditions and to get a better visual impression of the results when reading the plots.

Section 9.4 presents the statistical results for the complete measurements. All results shown will be discussed further in Section 10.

Table 10 describes the six different excerpts used throughout this section.

Nr.	Test period	Description
1	3920 - 3980 s	0 - 45 km/h, Low speed variation. Using Alg. 3
2	4110 - 4270 s	0 - 55 km/h, High speed variation. Using Alg. 3
3	5605 - 5680 s	0 - 40 km/h, Using Alg. 3
4	6400 - 6500 s	60 - 95 km/h, High speed. Using Alg. 3
5	4950 - 5010 s	0 - 45 km/h, Using Alg. 2
6	7000 - 7100 s	50 - 80 km/h, High Speed. Using Alg. 2

Table 10: Test result excerpts

9.1 Unaligned Measurements vs GPS reference

The following figures and tables present plots the results at their original sample rate vs the results from the reference system.

Figures 41 through 44 show plots for excerpts 1 to 4, using Algorithm 3. Figures 45 and 46 show plots for excerpts 5 and 6, using Algorithm 2.

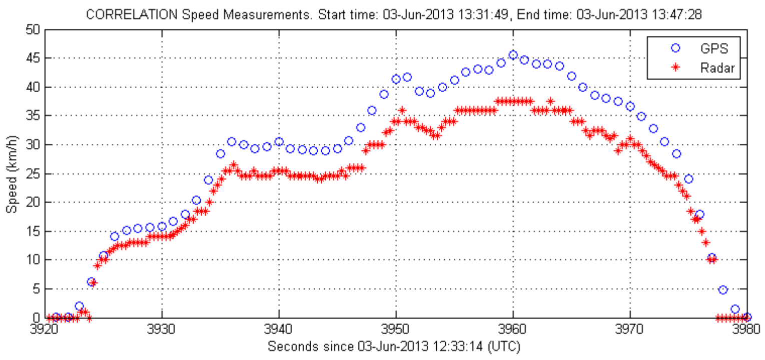


Figure 41: Measurement using Alg. 3 vs GPS Reference. Excerpt 1.

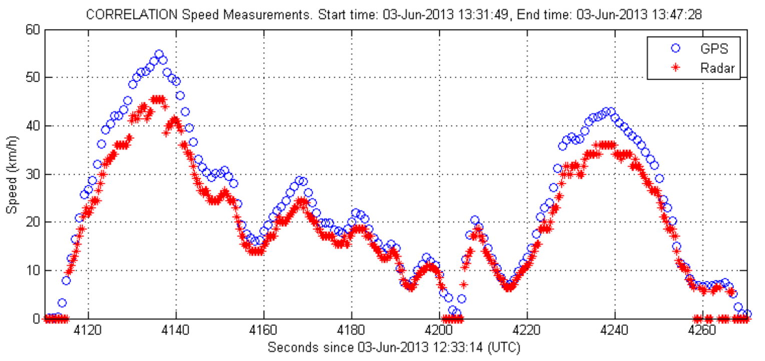


Figure 42: Measurement using Alg. 3 vs GPS Reference. Excerpt 2.

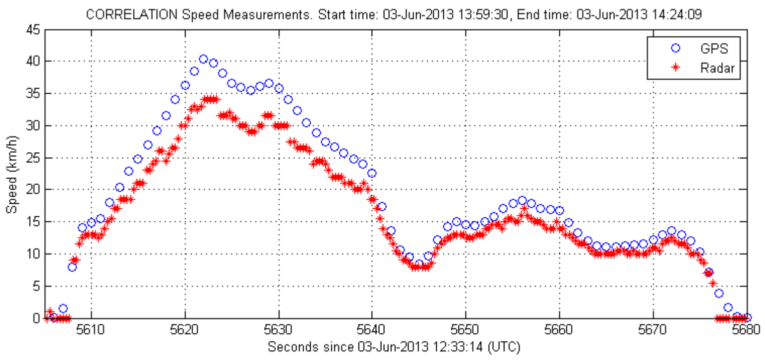


Figure 43: Measurement using Alg. 3 vs GPS Reference. Excerpt 3.

4

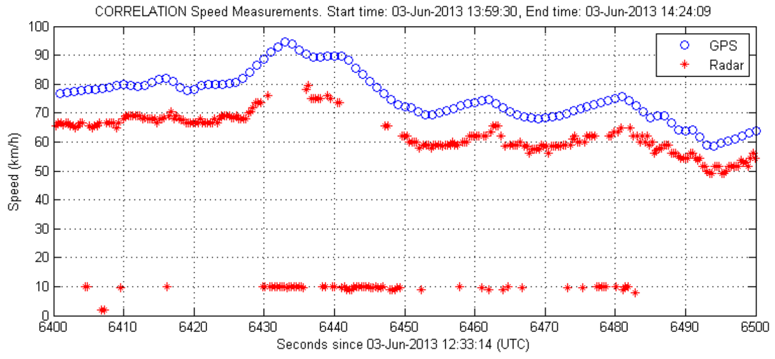


Figure 44: Measurement using Alg. 3 vs GPS Reference. Excerpt 4.

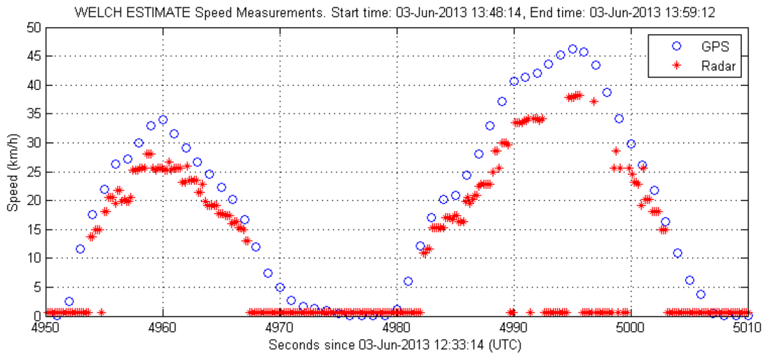


Figure 45: Measurement using Alg. 2 vs GPS Reference. Excerpt 5.

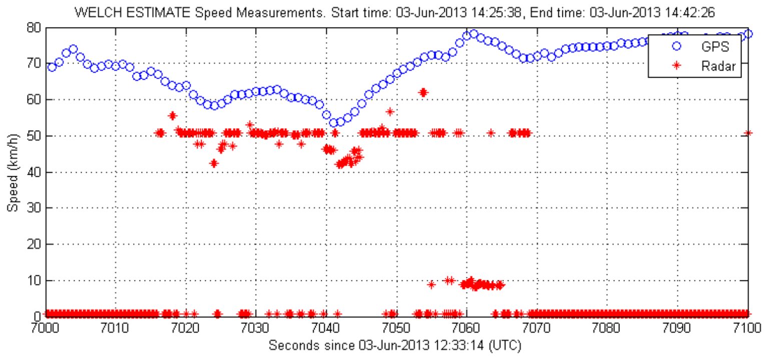


Figure 46: Measurement using Alg. 2 vs GPS Reference. Excerpt 6.

9.2 Aligned Measurements vs GPS reference

The following figures present plots of the down-sampled results vs the results from the reference system. In addition to plots of the speed measurements, histograms of the error in km/h and percentage are also shown. Only measurement values between 5 and 70 km/h are taken into account when the errors are calculated. Table 11 shows the statistical properties of the same excerpts.

Figures 47 through 50 show plots where the correlation algorithm was used. Figures 51 through 52 show plots where the direct Welch method was used to calculate the speed.

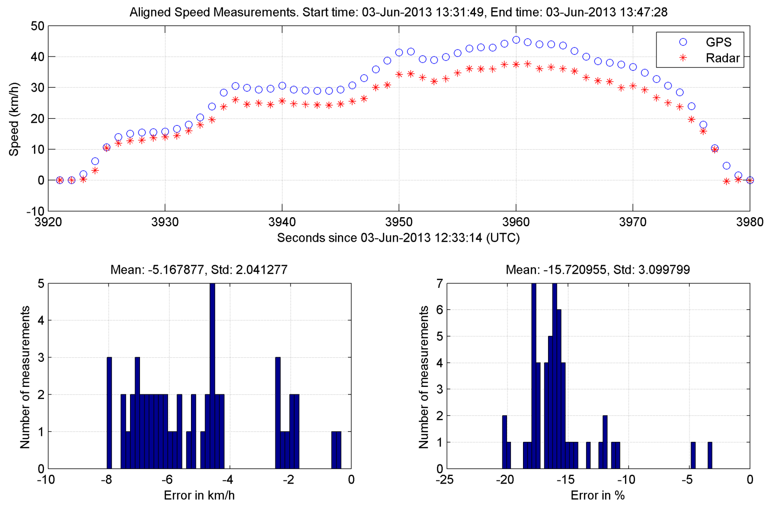


Figure 47: Aligned Measurement using Alg. 3 vs GPS Reference. Excerpt 1.

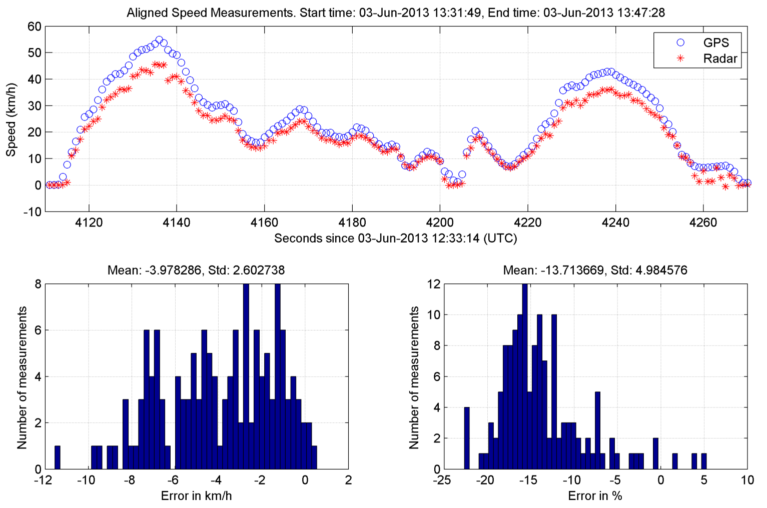


Figure 48: Aligned Measurement using Alg. 3 vs GPS Reference. Excerpt 2.

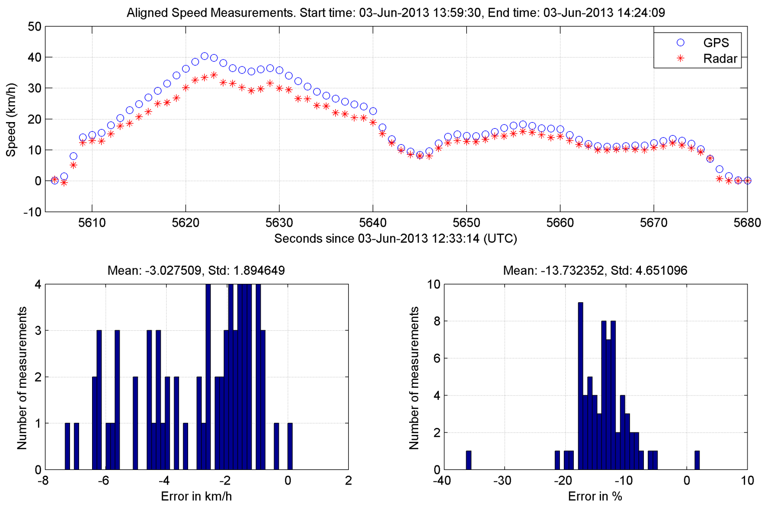


Figure 49: Aligned Measurement using Alg. 3 vs GPS Reference. Excerpt 3.

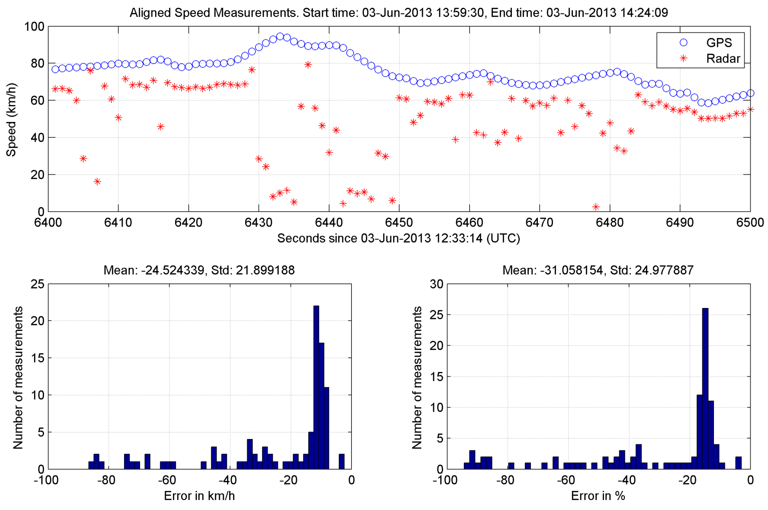


Figure 50: Aligned Measurement using Alg. 3 vs GPS Reference. Excerpt 4.

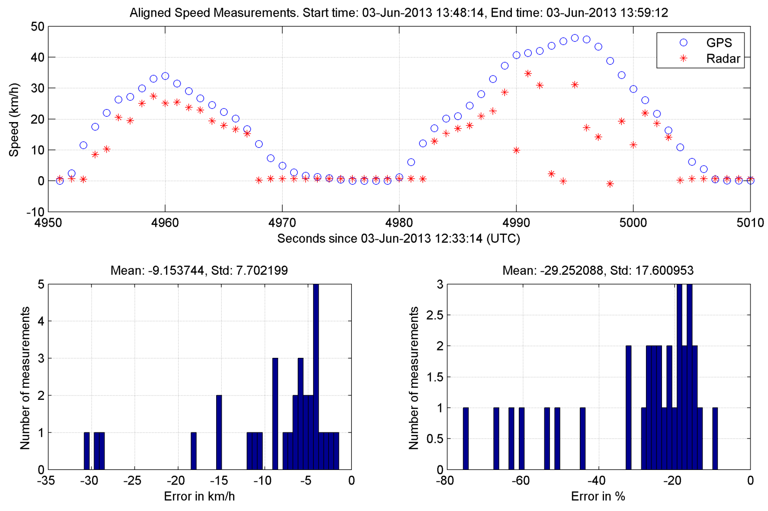


Figure 51: Aligned Measurement using Alg. 2 vs GPS Reference. Excerpt 5.

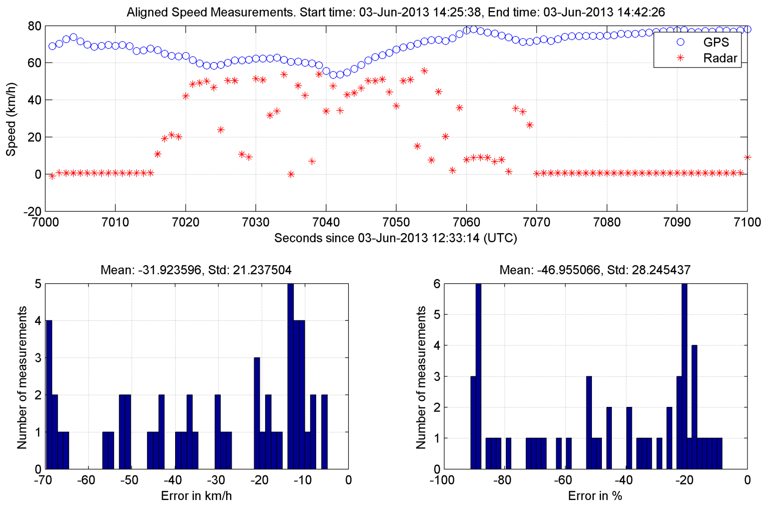


Figure 52: Aligned Measurement using Alg. 2 vs GPS Reference. Excerpt 6.

Excerpt Nr.	$\widehat{E}_{km/h}$	$\sigma_{km/h}$	$\widehat{E}_{\%}$	$\sigma_{\%}$
1	-5.17	2.04	-15.72	3.10
2	-3.98	2.60	-13.71	4.98
3	-3.02	1.89	-13.73	4.65
4	-24.52	21.90	-31.06	24.98
5	-9.15	7.70	-29.25	17.60
6	-31.92	21.24	-46.96	28.25

Table 11: Measurement Errors, Uncorrected Measurements Excerpts

9.3 Error Corrected Measurements vs GPS reference

The following figures present plots of the down-sampled and error corrected results vs the results from the reference system. An error correction factor of 1.18 was used. In addition to plots of the speed measurements, histograms of the error in km/h and percentage are also shown. Only measurement values between 5 and 70 km/h are taken into account when the errors are calculated. Table 12 shows the statistical properties of the same excerpts.

Figures 53 through 56 show plots where the correlation algorithm was used. Figures 57 through 58 show plots where the direct Welch method was used to calculate the speed.

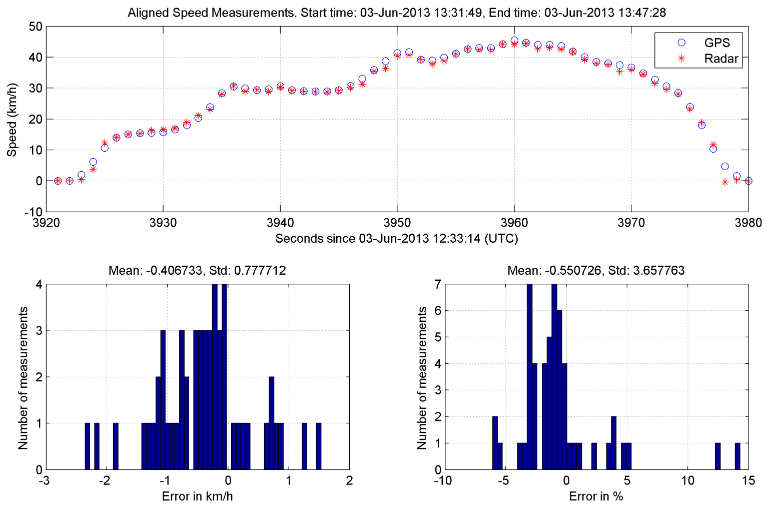


Figure 53: Error Corrected Measurement using Alg. 3 vs GPS Reference. Excerpt 1.

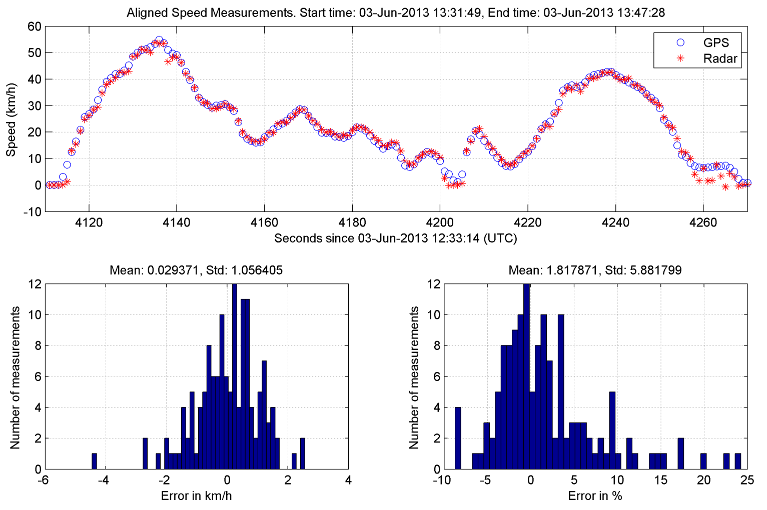


Figure 54: Error Corrected Measurement using Alg. 3 vs GPS Reference. Excerpt 2.

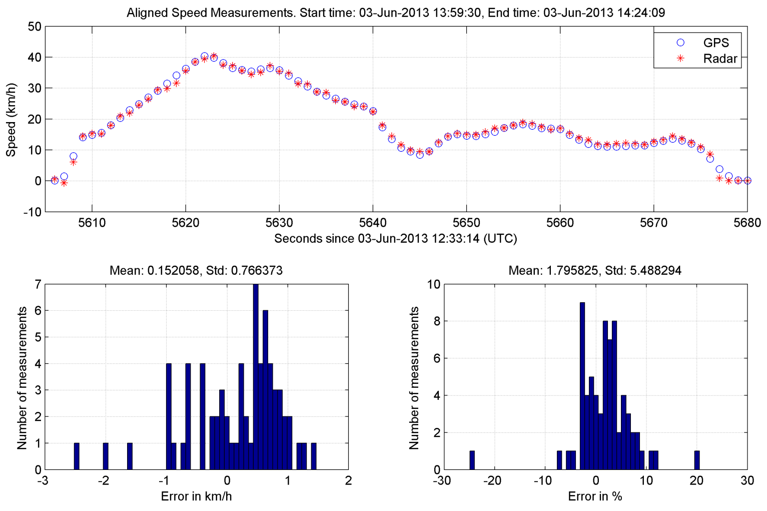


Figure 55: Error Corrected Measurement using Alg. 3 vs GPS Reference. Excerpt 3.

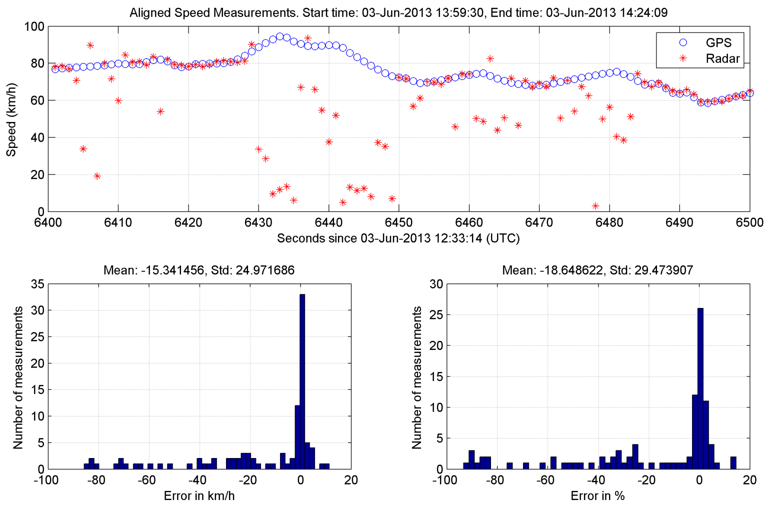


Figure 56: Error Corrected Measurement using Alg. 3 vs GPS Reference. Excerpt 4.

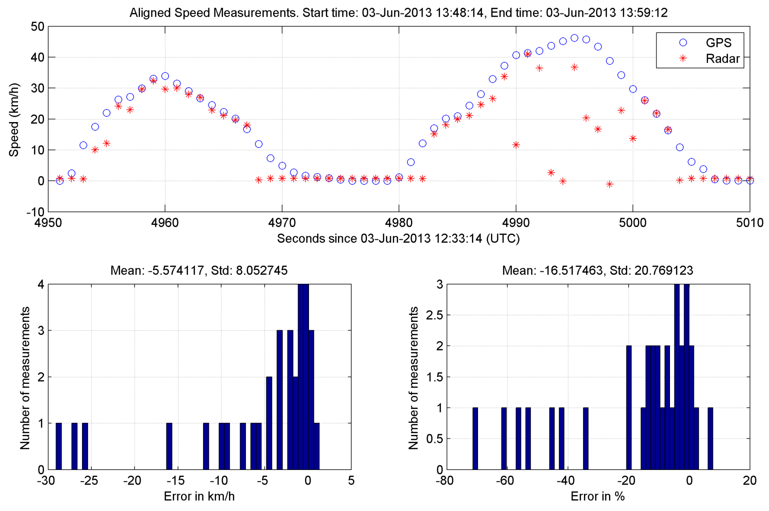


Figure 57: Error Corrected Measurement using Alg. 2 vs GPS Reference. Excerpt 5.

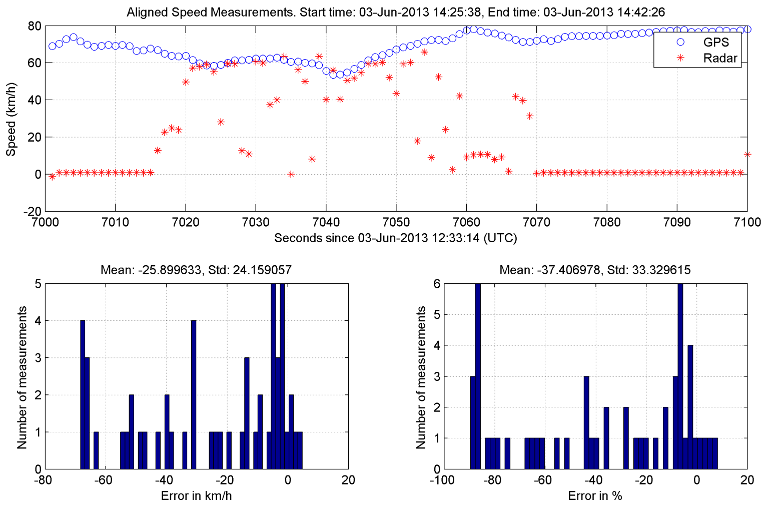


Figure 58: Error Corrected Measurement using Alg. 2 vs GPS Reference. Excerpt 6.

Excerpt Nr.	$\widehat{E}_{km/h}$	$\sigma_{km/h}$	$\widehat{E}_{\%}$	$\sigma_{\%}$
1	-0.41	0.77	-0.55	3.66
2	0.03	1.06	1.82	5.88
3	0.15	0.77	1.80	5.48
4	-15.34	24.97	-18.65	29.47
5	-5.57	8.05	-16.52	20.77
6	-25.90	24.16	-37.41	33.33

Table 12: Measurement Errors, Error Corrected Measurements Excerpts

9.4 Complete Measurement sequences vs GPS reference

The following figures present plots of the down-sampled results vs the results from the reference system for the complete four test runs described in Table 9. Both uncorrected and corrected measurements are shown, as well as histograms of the error in km/h and percentage. An error correction factor of 1.18 was used. Only measurement values between 5 and 70 km/h are taken into account when the errors are calculated. Table 13 shows the statistical properties of the same test runs.

Figures 59 and 60 show results from a 949 s continuous test run using Algorithm 3 to measure the speed, without and with error correction, respectively.

Figures 61 and 62 show results from a 1479 s continuous test run using Algorithm 3 to measure the speed, without and with error correction, respectively.

Figures 63 and 64 show results from a 658 s continuous test run using Algorithm 2 to measure the speed, without and with error correction, respectively.

Figures 65 and 66 show results from a 1008 s continuous test run using Algorithm 2 to measure the speed, without and with error correction, respectively.

Figure 67 shows a special case from test run 1 where only measurements between 20 and 70 km/h are taken into account when calculating the error, and an error correction factor of 1.20 was used. This case will be discussed in Section 10.4.3.

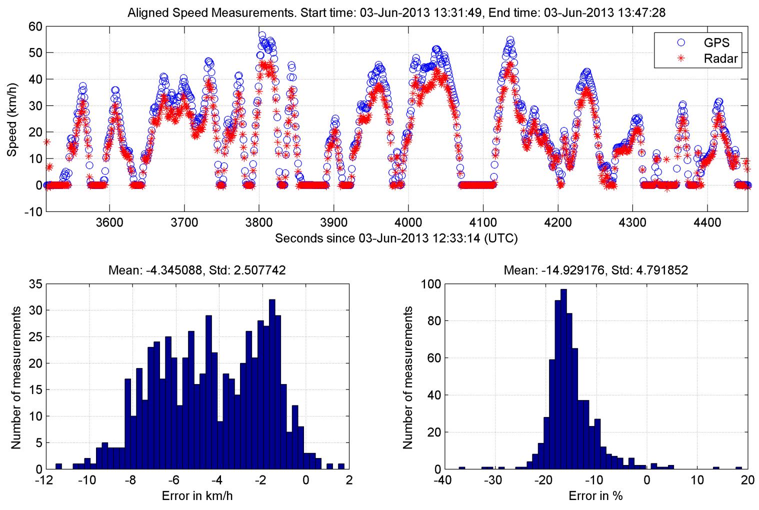


Figure 59: Aligned Measurement using Alg. 3 vs GPS Reference. Test run 1

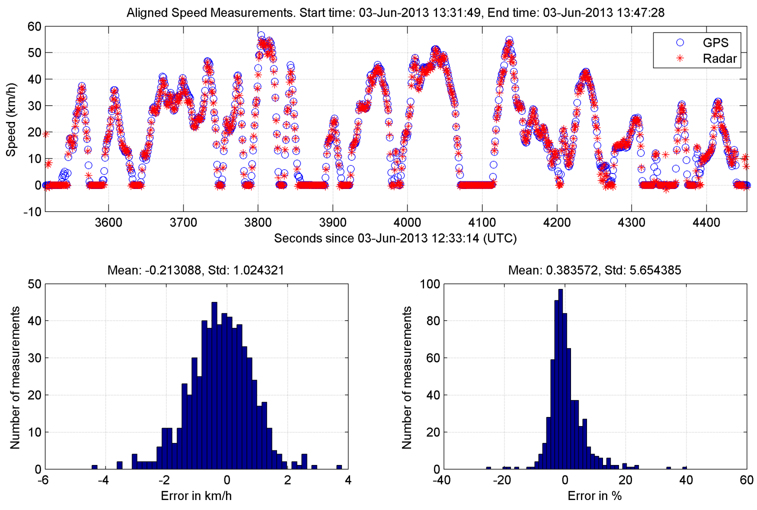


Figure 60: Error Corrected Measurement using Alg. 3 vs GPS Reference. Test run 1

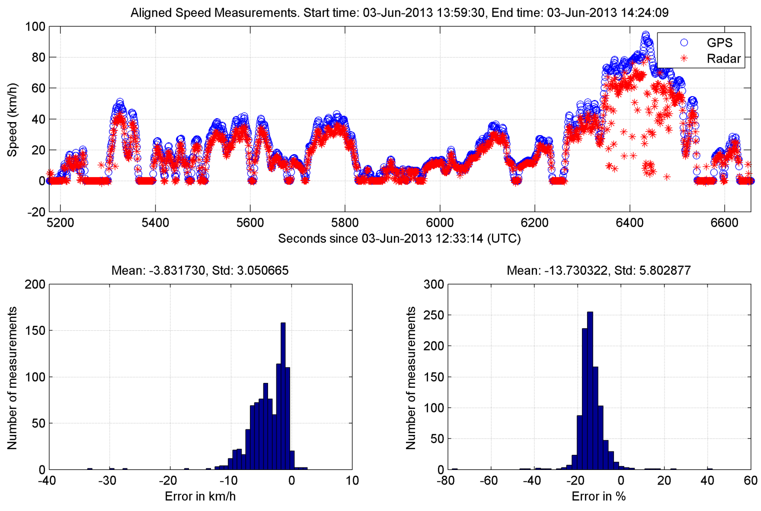


Figure 61: Aligned Measurement using Alg. 3 vs GPS Reference. Test run 2

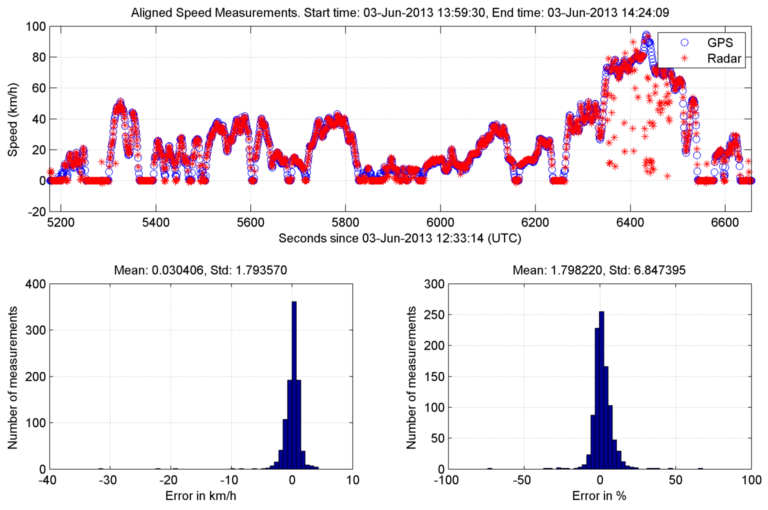


Figure 62: Error Corrected Measurement using Alg. 3 vs GPS Reference. Test run 2

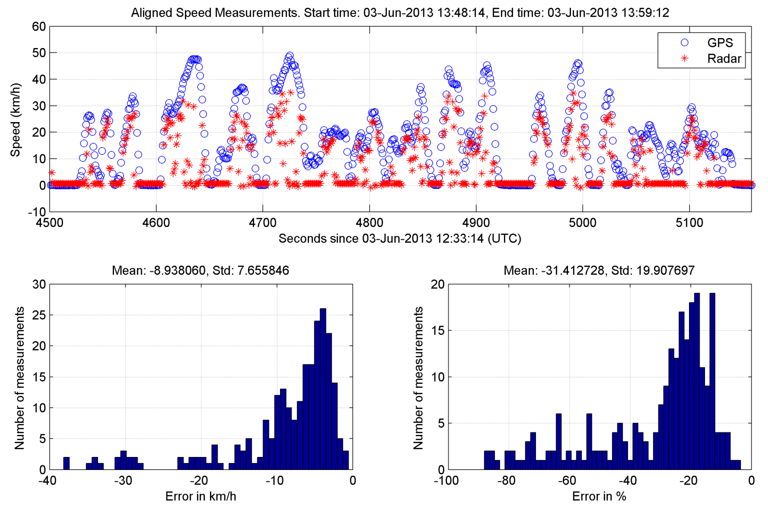


Figure 63: Aligned Measurement using Alg. 2 vs GPS Reference. Test run 3

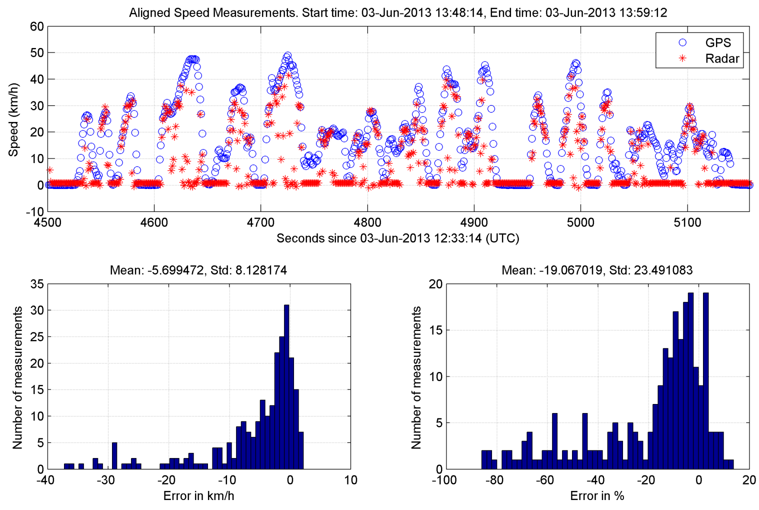


Figure 64: Error Corrected Measurement using Alg. 2 vs GPS Reference. Test run 3

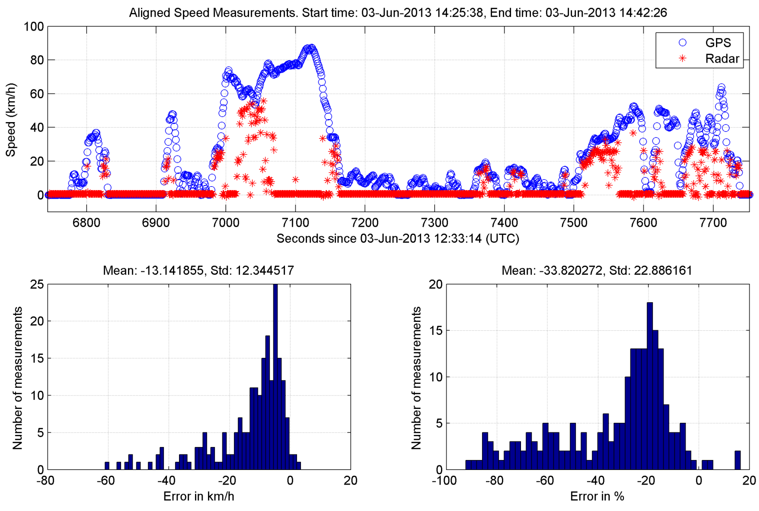


Figure 65: Aligned Measurement using Alg. 2 vs GPS Reference. Test run 4

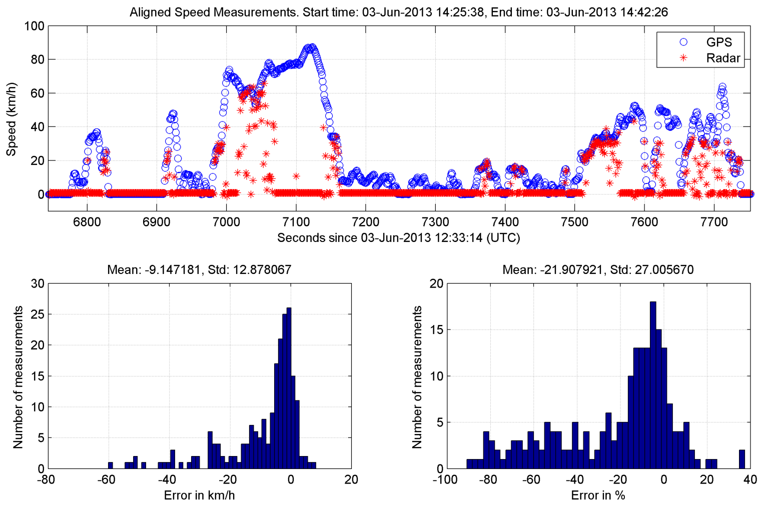


Figure 66: Error Corrected Measurement using Alg. 2 vs GPS Reference. Test run 4

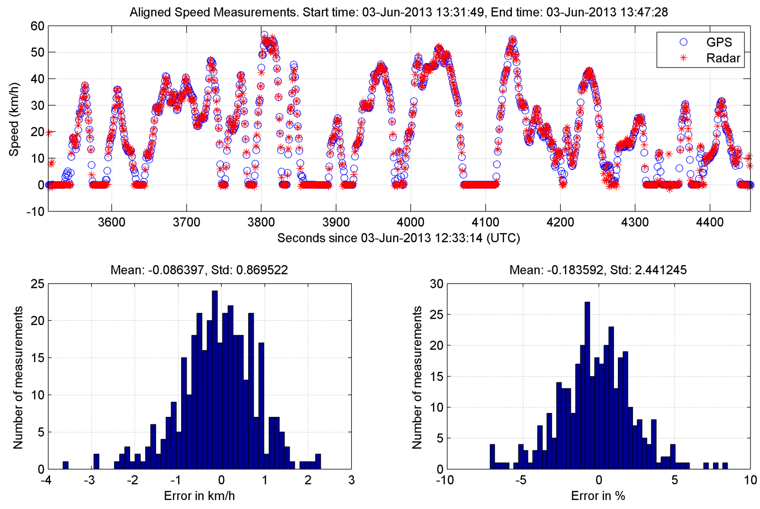


Figure 67: Error Corrected Measurement using Alg. 3 vs GPS Reference. Test run 1. Error correction factor = 1.20. Only measurements in the range of 20 to 70 km/h are taken into account.

Run Nr.	Corr.	$\widehat{E}_{km/h}$	$\sigma_{km/h}$	$\widehat{E}_{\%}$	$\sigma_{\%}$
1	1	-4.35	2.51	-14.93	4.79
1	1.18	-0.21	1.02	0.38	5.65
1 (20-70 km/h)	1.20	-0.09	0.87	-0.18	2.44
2	1	-3.83	3.05	-13.73	5.80
2	1.18	0.03	1.79	1.80	6.85
3	1	-8.94	7.66	-31.41	19.91
3	1.18	-5.70	8.13	-19.07	23.49
4	1	-13.14	12.34	-33.82	22.89
4	1.18	-9.15	12.88	-21.91	27.01

Table 13: Measurement Errors, Uncorrected and Corrected Full Test Runs.

9.5 Measurement Rates

The number of measurements per second outputted by the two algorithms were calculated by measuring the time between the measurement samples, and were as follows:

- Measurement rate for Algorithm 2: 5.211 Hz
- Measurement rate for Algorithm 3: 2.922 Hz

9.6 Measurement Resolutions

Algorithm 2
The resolution of this Algorithm 2 with respect to speed estimates are equivalent to the number of points in the Welch estimates. $2048/2 + 1 = 1025$ points in the one-sided spectra yields a resolution of $348\text{km/h}/1025 = 0.34\text{km/h}$, when the sampling frequency is 22050 Hz and the radar tilt angle is 45°.

Algorithm 3 The resolution of Algorithm 3 is identical to the maximum measurable speed divided by number of P_yy estimates: $250\text{km/h}/500 = 0.5$ km/h.

10 Discussion

This section will discuss the results that was presented in Section 9.

10.1 Measurable Range of Speeds

As can be clearly seen in Figure 44, the correlation algorithms struggles to measure speeds above approximately 70 km/h. The reason for this was not fully investigated in this thesis, but it can be assumed that the widening and flattering effect on the Doppler spectra when the speed is increased makes it hard to distinguish the Doppler spectra from the background noise. This, combined with the simple estimate described in Section 2.4 used when comparing the received spectrum, seams to yield poor measurement results when the speeds extend over 70 km/h.

As can also be seen in Figures 41 through 43, the coherence algorithm measurements seem to be unreliable for speed measurements under 5 km/h. This is due to the considerable amount of low frequency noise in the Doppler spectrum, and the subsequent filtering described in Section 4.3. Even though a FIR filter with 100 coefficients were used, the frequency response curve is not steep enough at the cut-off frequency to only reduce the noise at the lowest frequencies. Hence, the measurements at low speeds are effected, reducing the reliability.

However, in the range between 5 and 70 km/h, the measurements are reliable, and hence only the measurements that fall within this range were used when the error and accuracy of the measurements were analyzed.

10.2 Alignment of the Measurement Samples with Reference Samples

The figures in Section 9.2 show how the samples from the measurements were aligned with the samples from the GPS system. This shows that the interpolation/decimation method described in Section 8.1 worked as intended.

10.3 Statistics and Accuracy Before Error Correction

This part of the discussion is divided into four parts: The first discusses the results for measurement excerpts 1 to 3, as they have in common that no speeds over 70 km/h are encountered and that Algorithm 3 is used. The second part discusses Excerpt 4 which measures high speeds. Then Excerpts 4 and 5 are discussed, as they both use Algorithm 2. Finally, the results from the complete test runs are discussed.

10.3.1 Excerpts 1 to 3

When looking at the statistics presented in the figures and table in Section 9.2 for excerpts 1 to 3, the measurements have a mean percentage error in the range of -3.02 to -5.17 km/h. This implied that there was a constant error in the measurements, as is clearly shown in the figures. The standard deviations are relatively low, implying that the measurements are fairly accurate.

10.3.2 Excerpt 4

As described in Section 10.1, speeds over 70 km/h are not measured correctly. This is confirmed by the statistical data, which show a mean error of -24.52 km/h and a standard deviation of as much as 21.9 km/h. Hence, these results can not be seen as reliable.

10.3.3 Excerpts 4 and 6

The measurements using the strongest component of the Welch estimate to calculate the speed, exhibits a large mean errors, as well as large standard deviations, both in percentage and in km/h. As can be seen from Figure 51, the measurements are very unreliable even at fairly low speeds.

10.3.4 The Complete Test Runs

The uncorrected measurements presented in Section 9.4 show that the first two runs exhibit a relatively small standard deviation in km/h, but relatively large mean errors, confirming that a constant error is present.

10.4 Statistics and Accuracy After Error Correction

As described in Section 8.3, an error correction factor of 1.18 was used to compensate for the constant measurement error. When comparing the uncorrected results with the corrected results presented in Section 9.3, a significant improvement of the results can be observed.

10.4.1 Excerpts

The standard deviation drops down to 0.77 km/h for both excerpts 1 and 3, and the mean error for excerpt 2 is only 0.03 km/h. This confirms that a scaling factor can greatly improve the results, and that an inaccurate radar tilt angle can be compensated for by scaling the results.

The results from excerpts 4 confirms the unreliability when measuring speeds above 70 km/h and the poor measurements generated by Algorithm 2.

10.4.2 The Complete Test Runs

When looking at the results from the complete test runs presented in Table 13, a similar improvement can be observed. Run 1 has mean errors of only -0.21 km/h and 0.38 %, and a standard deviation of 1.02 km/h.

Run 2 has an even lower mean error of just 0.03 km/h. A slightly higher mean error in percentage results from the measurements in the last section where the speeds approaches 70 km/h, as can be seen in Figure 61.

When keeping in mind that the resolution used in the correlation algorithm is 0.5 km/h, these results seem very reliable.

The results from run 3 and 4 again confirms that Algorithm 2 is not reliable, even at low speeds. Hence, Algorithm 3 should be used as the preferred method for measuring the speeds.

10.4.3 Unlinearity of Mean Error

To show that the error in the results are not completely linear, a special case was added where only measurements within the range of 20 - 70 km/h were taken into account. The results from this case are shown in Figure 67 and in line three of Table 13.

Here, it can be seen that using an error correction factor of 1.20 further improves the accuracy of the results. The mean errors are now -0.09 km/h and -0.18 %, with low standard deviations of 0.87 and 2.44 in km/h and %, respectively.

More accurate estimations of the theoretical power spectra used in the correlation algorithm should remove some of this unlinearity.

10.5 Measurement Rates and Resolution

The measurement rates of 5.211 and 2.922 Hz are both within the specifications set in Section 1.2.2. The rates can be further improved by reducing the number of samples used in the Welch estimate, and by comparing the measured power spectra to a smaller amount of theoretical spectra, i.e. comparing to spectra close to the previous estimated speed, or reducing the maximum measurable speed.

The resolution of the measurements were 0.34 km/h for Algorithm 2 and 0.5 km/h for Algorithm 3, as shown in Section 9.6. These fall within the set specifications, but can also be improved. Choosing a lower sample rate will increase the resolution of the Welch estimate, but lower the maximal measurable speed. Increasing the number of theoretical spectra used in the correlation algorithm will also improve the resolution, but will require more computational power, i.e. more time, reducing the measurement rate unless further improvements of the algorithm are implemented.

10.6 Comparing the Results with Previous Solutions

The speed measurements are much more accurate than the output from a speedometer, as can be seen by comparing the corrected results with Equation 1.1 which states that the error of a speedometer is less than 10 % of the true speed + 4 km/h.

Compared to a GPS system, the accuracy is as described in the results, as a GPS system was used as the reference.

The earlier Doppler radar that were presented in Section 1.4 reports an accurateness in the range of 0.5 to 2.0 %. The first approach reports 0.5 to 2.0 % accuracy, but the vehicle in question runs at a constant speed, and the system outputs values at 10 second intervals, making it hard to compare to the system described in this thesis.

The last approach uses two radar sensors to effectively eliminate the impact of the pitch angle of the vehicle, thereby improving the results and obtaining a standard deviation in percentage points of 0.6 %. However, this system is more sophisticated, and a direct comparison is therefore unfair to the system implemented in this thesis. But with more work invested in solving the problems with the limited measurable range, and in improving the theoretical estimates of the doppler spectra, the performance of the implemented system should improve significantly.

11 Concluding Remarks

A system for measuring the speed of vehicles using Doppler radar was developed and implemented. The system can make use of two different algorithms to measure the speed: Selecting the strongest frequency component of a Welch power density spectrum estimate, or correlating the Welch estimate with pre-estimated theoretical Doppler spectra for different speeds to find the best possible match, and hence the speed of the vehicle.

The system was tested using a test vehicle and a GPS reference system. The results show that a mean error as low as 0.03 km/h and -0.18 % was observed, with standard deviations of 0.87 km/h and 2.44 %, during test runs of 949 and 1479 seconds. These results were obtained using the correlation algorithm for speed measurement (Algorithm 3), which was proven to be the most reliable of the algorithms that were tested. It was also found that an error in the radar tilt angle can be compensated for using an error correction factor.

Measurement rates of 2.911 Hz were obtained using the correlation algorithm, and 5.211 Hz using the direct Welch method, both well within the set specifications for the system.

11.1 Future Work

Future work have to be invested to improve the measurable range of speeds, and some recommendations to areas that should be explored are mentioned in the following paragraphs.

Improving the Measurable Range
Research should be done on the possibility of using an analog filter to amplify and filter the IF signal before it is fed to the AD converted in the CODEC. This to remove as much noise as possible before the signal is digitized, and hence make it easier to measure higher speeds where the Doppler spectrum components are weaker and low speeds where large amounts of noise distorts the signal.

Improving the Accuracy
The most obvious way to improve the accuracy is to improve the theoretical Doppler spectrum estimates, P_{yy}, used by the correlation algorithm. This is the area that mostly affects the outcome of the measurements.

Averaging subsequent measurements could also be a way to improve the accuracy, as well as discarding measurements that are illogical, e.g. extreme speed changes that are impossible for a car.

References

[1] W. Kleinhempel, W. Stammler, and D. Bergmann. Radar signal processing for vehicle speed measurements. In *EUSIPCO'92. Proceedings of*, pages 1533–1536, 1992.

[2] John G. Proakis and Dimitris G. Manolakis. *Digital signal processing*. Pearson Prentice Hall, Upper Saddle River, N.J., 4th edition, 2007.

[3] ARM Ltd. CMSIS - Cortex Microcontroller Software Interface Standard. `http://www.arm.com/products/processors/cortex-m/cortex-microcontroller-software-interface-standard.php`. Accessed: 08 June 2013.

[4] William Harris. How Speedometers Work. `http://auto.howstuffworks.com/car-driving-safety/safety-regulatory-devices/speedometer.htm`. Accessed: 08 June 2013.

[5] European Council. Directive 75/443/EEC of 26 June 1975 on the approximation of the laws of the Member States relating to the reverse and speedometer equipment of motor vehicles. OJ L 196, pp. 1-5, 26 June 1975.

[6] The Navigation Center of Excellence. *NAVSTAR GPS User Equipment Introduction*. United States Government, 1996.

[7] Tom Chalko. Estimating Accuracy of GPS Doppler Speed Measurement using Speed Dilution of Precision (SDOP) parameter, 2009.

[8] S. S. Stuchly, A. Thansandote, J. Mladek, and J. S. Townsend. A Doppler radar velocity meter for agricultural tractors. *Vehicular Technology, IEEE Transactions on*, 27(1):24–30, 1978.

[9] Masao Kodera, Seishin Mikami, Kunihiko Sasaki, and Jyunshi Utsu. Doppler radar speed detecting method and apparatus therefor, 1991.

[10] W. Kleinhempel, D. Bergmann, and W. Stammler. Speed measure of vehicles with on-board Doppler radar. In *Radar 92. International Conference*, pages 284–287, 1992.

[11] W. Kleinhempel. Automobile Doppler speedometer. In *Vehicle Navigation and Information Systems Conference, 1993., Proceedings of the IEEE-IEE*, pages 509–512, 1993.

[12] Patrick D. L. Beasley. Doppler radar speed sensor, 1993.

[13] Merrill I. Skolnik. *Introduction to Radar Systems*. McGraw-Hill, Inc., 1981.

[14] Christian Hülsmeier. Hertzian-wave Projecting and Receiving Apparatus Adapted to Indicate or Give Warning of the Presence of a Metallic Body, Such as a Ship or a Train, in the Line of Projection of Such Waves, 1904.

[15] Microwave Solutions Ltd. Application Note: Using Microwave Solutions Ltd Motion Detector Snits, 2012.

[16] F. B. Berger. The Nature of Doppler Velocity Measurement. *Aeronautical and Navigational Electronics, IRE Transactions on*, ANE-4(3):103–112, 1957.

[17] F. Placentino, F. Alimenti, A. Battistini, W. Bernardini, P. Mezzanotte, V. Palazzari, S. Leone, A. Scarponi, N. Porzi, M. Comez, and L. Roselli. Measurements of length and velocity of vehicles with a low cost sensor radar Doppler operating at 24GHz. In *Advances in Sensors and Interface, 2007. IWASI 2007. 2nd International Workshop on*, pages 1–5, 2007.

[18] Ville Viikari, Timo Varpula, and Mikko Kantanen. Automotive Radar Technology for Detecting Road Conditions. Backscattering Properties of Dry, Wet, and Icy Asphalt. In *EuRAD, 2008 5th European Radar Conference*, pages 276–279, 2008.

[19] Adrian K. Fung. *Microwave Scattering and Emission Models and Their Applications.* Boston: Artech House, 1994.

[20] Microwave Solutions Ltd. Datasheet: K-Band Doppler Motion Detector Units, Model Numbers MDU2400/2410, 2013.

[21] Peter D. Welch. The use of fast Fourier transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms. *Audio and Electroacoustics, IEEE Transactions on*, 15(2):70–73, 1967.

[22] NXP Semiconductors. LPC43xx ARM Cortex-M4/M0 dual-core microcontroller User Manual, 2012.

[23] KEIL Tools by ARM Ltd. MCB4300 Evaluation Board. `http://www.keil.com/mcb4300/`. Accessed: 08 June 2013.

[24] NXP Semiconductors. Datasheet: UDA1380 Stereo audio coder-decoder for MD, CD and MP3, 2010.

[25] Philips Semiconductors. Datasheet: I2S bus specificaion, 1996.

[26] Tektronix Inc. Primer: Understanding FFT Overlap Processing Fundamentals, 2009.

[27] LPCOpen Platform for NCP LPC Microcontrollers, Online Documentation. `http://docs.lpcware.com/lpcopen/v1.03/`. Accessed: 08 June 2013.

A Source Code

A.1 Main Application

```c
#include "app.h"
#include "app_config.h"
#include "board.h"
#include "chip.h"
#include "spectrum.h"
#include "speed.h"
#include "memory_map.h"
#include "dsp_funcs.h"
#include "uda1380_mcb4300.h"
#include "display_mcb4300.h"
#include "sdmmc_mcb4300.h"
#include "rtc.h"
#include <stdlib.h>

// Text color for menu
#define MENU_TEXT_COLOR Cyan
#define MENU_BACK_COLOR Black
#define SAVE_TEXT_COLOR Yellow

// Menu line numbers for different items
// Main menu
#define SPECTRUM 2
#define SPEED 1
#define RECORD 3

// Speed menu
#define PERIOD 1
#define WELCH 2
#define COHERE 3

// Total items in menu
#define MAIN_MENU_ITEMS 3
#define SPEED_MENU_ITEMS 3

// sprintf buffer size
#define BUF_SIZE 200

// Strings to print.
static const char arrow_s[] =       "->";
static const char arrow_clr_s[] =   "  ";
static const char angle_s[] =       "\fPlease enter radar tilt angle (in degrees),\n"
                                    "where the horizontal axis is 0 degrees:";
static const char recording_s[] =   "\fRecording...\nPress to stop.\n\n";
static const char save_file_s[] =   "\fSave file to *.WAV:\n"
                                    "- Max name length: 8 characters\n"
                                    "- Joystick up/down to select characters\n"
                                    "- Joystick right to enter next character\n"
                                    "- Press to save\n\n\n\n";
static const char done_s[] =        "Done.\nPress to return.\n";
static const char save_s[] =        "\fSaving file, please wait..";
static const char rtc_init_s[] =    "\fInitializing RTC (might take few seconds)...";

// File ending to use when saving files
static const char file_ext[] =      "WAV";

// Main Menu
static const char main_menu_s[]  =  "SELECT OPERATION\n"
                                    "   Speed Measurement\n"
                                    "   Spectrum Analyzer\n"
                                    "   Record Raw Data Sequence";
// Speed Menu
static const char speed_menu_s[]  = "SELECT ALGORITHM TYPE\n"
                                    "   Single Periodogram\n"
                                    "   Welch Estimation\n"
                                    "   Spectral Coherence";

// List of menus
```

```c
static const char* menus[2] = {main_menu_s, speed_menu_s};


/*---------------------------------------------------------------------------
  App function
 *---------------------------------------------------------------------------*/
void app_system(void){
    // Shared variables from uda1380_mcb4300.c
    extern volatile int play_done;
    extern volatile int rec_done;

    // Angle variable
    static float32_t angle;

    // Local variables
    static int init = 0;
    static int i;
    static int a = 45;
    static char buf[BUF_SIZE];
    static char fname[16];
    static char c;
    static int menu_nr = 0;
    static int item_nr = 1;
    static int nr_of_items = 0;
    static RTC rtctime;
    static int count = 0;
    static uint8_t blank = 0;
    static uint8_t joy = NO_BUTTON_PRESSED;
    static uint8_t joyMask = NO_BUTTON_PRESSED;

    // The different menus
    static enum{main_menu, speed_menu} Menu = main_menu;

    // Application states
    static enum{set_angle, show_clock, set_clock, print_menu, menu_active,
                speed_active, spectrum_active, rec_active, play_active,
                save_file, load_file, wait_for_back} State = set_angle;

    // Application state machine;
    switch (State) {
        case set_angle:
            // Initial state to set the tilt angle of the radar. Prompts the
            // user to enter the correct angle in degrees
            if(!init){
                init = 1;
                display_set_font_size(SMALL_FONT);
                display_set_text_color(MENU_TEXT_COLOR);
                display_set_back_color(MENU_BACK_COLOR);
                display_clear(MENU_BACK_COLOR);
                display_print_string(0,0,angle_s);
                sprintf(buf,"%02d\n",a);
                display_print_string(4,1,buf);
            }
            // Read joystick input
            joyMask = Joystick_GetStatus();
            if(joyMask != joy){
                joy = joyMask;
                if(joy & JOY_RIGHT){
                    // Decrease value
                    if (a == 90) a = 0;
                    else a--;
                    sprintf(buf,"%02d\n",a);
                    display_print_string(4,1,buf);
                }
                else if(joy & JOY_LEFT){
                    // Increase value
                    if (a == 0) a = 90;
                    else a++;
                    sprintf(buf,"%02d\n",a);
                    display_print_string(4,1,buf);
                }
                else if(joy & JOY_PRESS){
                    // Save value
                    angle = ((float)a*PI)/180.0f;
                    count = 0;
                    display_clear(MENU_BACK_COLOR);
                    State = show_clock;
                }
            }// end if (joyMask != joy)
            break; // End case set_angle

        case show_clock:
            // Checks the rurrent RTC time, and prompts the user to
            // verify or change the time and date.
            if(count == 0){
                rtc_gettime(&rtctime);
                sprintf(buf,"Current time: %02d:%02d:%02d\n"
                            "Current date: %04d/%02d/%02d\n\n"
                            "Press to verify, Up to change.",
                            rtctime.hour,rtctime.min,rtctime.sec,
                            rtctime.year,rtctime.month,rtctime.mday);
                display_print_string(0,0,buf);
            }
            count++;
                if(count == 10) count = 0;

            joyMask = Joystick_GetStatus();
            if(joyMask != joy){
                joy = joyMask;
                if(joy & JOY_LEFT){
                    // Time is not correct...
                    // Initialize RTC
                    display_print_string(0,0,rtc_init_s);
                    Chip_RTC_Init();

                    // Set default time and date to make it easier to set a new one
                    if (rtctime.year < 2013){
                        rtctime.sec = 0;
                        rtctime.min = 0;
                        rtctime.hour = 0;
                        rtctime.wday = 3;
                        rtctime.mday = 30;
                        rtctime.month = 5;
                        rtctime.year = 2013;
                    }
                    snprintf(buf,BUF_SIZE,"\fPlease set correct time:\n\n"
                       "New time: %02d:%02d:%02d\n"
                       "New date: %04d/%02d/%02d\n\n"
                       "- Up/down to change value.\n"
                       "- Left/right to choose item.\n"
                       "- Press to verify and set time.\n",
                        rtctime.hour,rtctime.min,rtctime.sec,
                        rtctime.year,rtctime.month,rtctime.mday);
                    display_print_string(0,0,buf);
                    count = 0;
                    State = set_clock;
                }
                else if(joy & JOY_PRESS){
                    // Time is correct
                    // Initialize SDMMC Peripheral
                    sdmmc_init();
                    count = 0;
                    State = print_menu;
                }
            }// end if (joyMask != joy)
            break; // end case show_clock

        case set_clock:
            // The user is promted to enter the correct time and date
            // for the system.
            joyMask = Joystick_GetStatus();
            if(joyMask != joy){
                joy = joyMask;
                if(joy & JOY_UP){
                    // Next element
                    if(i < 5) i++;
                }
                else if(joy & JOY_DOWN){
                    // Previous element
                    if(i > 0) i--;
                }
                else if(joy & JOY_RIGHT){
                    // Decrement value
                    switch(i){
                        case 0:
                            rtctime.hour--;
                            if(rtctime.hour > 23) rtctime.hour = 23;
                            break;
                        case 1:
                            rtctime.min--;
                            if(rtctime.min > 59) rtctime.min = 59;
                            break;
                        case 2:
                            rtctime.sec--;
                            if(rtctime.sec > 59) rtctime.sec = 59;
                            break;
                        case 3:
                            rtctime.year--;
                            if(rtctime.year < 1) rtctime.year = 4095;
                            break;
                        case 4:
                            rtctime.month--;
                            if(rtctime.month < 1) rtctime.month = 12;
                            break;
                        case 5:
                            rtctime.mday--;
                            if(rtctime.mday < 1) rtctime.mday = 31;
                            break;
                    }
                }
                else if(joy & JOY_LEFT){
                    // Increment Value
                    switch(i){
                        case 0:
                            rtctime.hour++;
                            if(rtctime.hour > 23) rtctime.hour = 0;
                            break;
                        case 1:
                            rtctime.min++;
                            if(rtctime.min > 59) rtctime.min = 0;
                            break;
                        case 2:
                            rtctime.sec++;
                            if(rtctime.sec > 59) rtctime.sec = 0;
                            break;
                        case 3:
                            rtctime.year++;
                            if(rtctime.year > 4096) rtctime.year = 1;
                            break;
                        case 4:
                            rtctime.month++;
                            if(rtctime.month > 12) rtctime.month = 1;
                            break;
                        case 5:
                            rtctime.mday++;
                            if(rtctime.mday > 31) rtctime.mday = 1;
                            break;
                    }
                }
                else if(joy & JOY_PRESS){
                    // Set new time and enable RTC counter
                    rtc_settime(&rtctime);
                    Chip_RTC_Enable(ENABLE);
                    // Initialize SDMMC Peripheral
                    sdmmc_init();
                    State = print_menu;
                }
            }// end if (joyMask != joy)

            count ++;
            if (count == 5){ // @ 0.5 seconds
                count = 0;
```

```
                if(blank){
                    // Clear the active time/date value to make it "blink"
                    switch(i){
                        case 0:
                            display_print_string(2,10,arrow_clr_s);
                            break;
                        case 1:
                            display_print_string(2,13,arrow_clr_s);
                            break;
                        case 2:
                            display_print_string(2,16,arrow_clr_s);
                            break;
                        case 3:
                            display_print_string(3,10,arrow_clr_s);
                            display_print_string(3,12,arrow_clr_s);
                            break;
                        case 4:
                            display_print_string(3,15,arrow_clr_s);
                            break;
                        case 5:
                            display_print_string(3,18,arrow_clr_s);
                            break;
                    }
                }
                else{
                    // Update the display with the new time
                    sprintf(buf,"New time: %02d:%02d:%02d\n"
                                "New date: %04d/%02d/%02d\n",
                                rtctime.hour,rtctime.min,rtctime.sec,
                                rtctime.year,rtctime.month,rtctime.mday);
                    display_print_string(2,0,buf);
                }
                blank = !blank;
            }// end if(count == 5)
            break; // end case set_clock

        case print_menu:
            // Clear the screen and print the menu selected by the Menu variable.
            switch (Menu) {
                case main_menu:
                    menu_nr = 0;
                    nr_of_items = MAIN_MENU_ITEMS;
                    break;
                case speed_menu:
                    menu_nr = 1;
                    nr_of_items = SPEED_MENU_ITEMS;
                    break;
            }// end switch(menu)
            display_set_font_size(SMALL_FONT);
            display_set_text_color(MENU_TEXT_COLOR);
            display_set_back_color(MENU_BACK_COLOR);
            display_clear(MENU_BACK_COLOR);
            display_print_string(0,0,menus[menu_nr]);
            display_print_string(item_nr,0,arrow_s);
            State = menu_active;
            break;//end case print_menu

        case menu_active:
            // Print "->" at selected menu item. Go to correct state when an item is
                 selected
            joyMask = Joystick_GetStatus();
            if(joyMask != joy){
                joy = joyMask;
                if(joy & JOY_LEFT){
                    // Go to previous item
                    display_print_string(item_nr,0,arrow_clr_s);
                    if(item_nr == 1)
                        item_nr = nr_of_items;
                    else
                        item_nr--;
                    display_print_string(item_nr,0,arrow_s);
                }
                else if(joy & JOY_RIGHT){
                    // Go to next item
                    display_print_string(item_nr,0,arrow_clr_s);
                    if(item_nr == nr_of_items)
                        item_nr = 1;
                    else
                        item_nr++;
                    display_print_string(item_nr,0,arrow_s);
                }
                else if(joy & JOY_PRESS){
                    switch(Menu){
                        // Enter correct state according to the active menu
                        case main_menu:
                            switch(item_nr){
                                case RECORD:
                                    // Start recording and go to rec_active state
                                    display_print_string(0,0,recording_s);
                                    uda1380_start_rec((REC_END_ADDR-REC_START_ADDR)/4,
                                        (int32_t * )REC_START_ADDR);
                                    State = rec_active;
                                    break;

                                case SPECTRUM:
                                    // Initialize the spectrum analysator.
                                    spectrum_init(SAMPLE_RATE, FFT_POINTS/2+1);
                                    State = spectrum_active;
                                    break;

                                case SPEED:
                                    // Set the Speed menu as active menu
                                    Menu = speed_menu;
                                    item_nr = 1;
                                    State = print_menu;
                                    break;
                            }// end switch(item_nr)
                            break; // end case main_menu

                        case speed_menu:
                            switch(item_nr){
                                case PERIOD:
                                    speed_init(angle,SPEED_PERIODOGRAM);
                                    State = speed_active;
                                    break;

                                case WELCH:
                                    speed_init(angle,SPEED_WELCH);
                                    State = speed_active;
                                    break;

                                case COHERE:
                                    speed_init(angle,SPEED_COHERENCE);
                                    State = speed_active;
                                    break;
                            }// end switch(item_nr)
                            break; // end case speed_menu
                    }// end switch(Menu)
                }// end if (joy & JOY_PRESS)
            }// end if (joyMask != joy)
            break;// end case menu_active

        case speed_active:
            // Wait for the user to press the joystick, ending
            // the speed measurement
            joyMask = Joystick_GetStatus();
            if(joyMask != joy){
                joy = joyMask;
                if(joy & JOY_PRESS){
                    display_set_font_size(SMALL_FONT);
                    display_print_string(0,0,save_s);
                    // Deinitialize speed module
                    speed_deinit();
                    display_print_string(2,0,done_s);
                    Menu = main_menu;
                    item_nr = SPEED;
                    State = wait_for_back;
                }
            }
            break; // end case speed_active

        case spectrum_active:
            // Wait for user to press joystick, deinitialize the spectrum analysator
            // and return to main menu.
            joyMask = Joystick_GetStatus();
            if(joyMask != joy){
                joy = joyMask;
                if(joy & JOY_PRESS){
                    spectrum_uninit();
                    State = print_menu;
                }
            }
            break;// end case spectrum_active

        case rec_active:
            // Wait for the recording to finish or for the user to press the joystick
            // before going to the save_file state
            joyMask = Joystick_GetStatus();
            if(joyMask != joy){
                joy = joyMask;
                if(joy & JOY_PRESS){
                    rec_done = 1;
                }
            }
            if(rec_done){
                rec_done = 0;
                uda1380_stop_rec();
                i = 0;
                c = 'A';
                display_print_string(0,0,save_file_s);
                display_set_text_color(SAVE_TEXT_COLOR);
                display_print_char(6,i,c);
                State = save_file;
            }
            break;// end case rec_active

        case save_file:
            // Get the file-name to use from user input, then add the
            // file extention and save the file, before going to the
            // wait_for_back state.
            joyMask = Joystick_GetStatus();
            if(joyMask != joy){
                joy = joyMask;
                if(joy & JOY_UP){
                    display_set_text_color(MENU_TEXT_COLOR);
                    display_print_char(6,i,c);
                    fname[i] = c;
                    if(i<7) i++;
                    display_set_text_color(SAVE_TEXT_COLOR);
                    display_print_char(6,i,c);
                }
                else if(joy & JOY_DOWN){
                    //if(i>0) i--;
                }
                else if(joy & JOY_RIGHT){
                    if (c == '9') c = 'A';
                    else if (c == 'Z') c = '0';
                    else c++;
                    display_set_text_color(SAVE_TEXT_COLOR);
                    display_print_char(6,i,c);
                }
                else if(joy & JOY_LEFT){
                    if (c == '0') c = 'Z';
                    else if (c == 'A') c = '9';
                    else c--;
                    display_set_text_color(SAVE_TEXT_COLOR);
                    display_print_char(6,i,c);
                }
                else if(joy & JOY_PRESS){
                    display_set_text_color(MENU_TEXT_COLOR);
                    display_print_char(6,i,c);
                    fname[i] = c;
                    fname[i+1] = '.';
                    fname[i+2] = file_ext[0];
                    fname[i+3] = file_ext[1];
                    fname[i+4] = file_ext[2];
                    fname[i+5] = '\0';
                    sprintf(buf,"\fSaving file \"%s\"...",fname);
                    display_print_string(0,0,buf);
                    sdmmc_save_wav(fname, (int32_t * )REC_START_ADDR, uda1380_stop_rec
                        ());
                    display_print_string(1,0,done_s);
                    State = wait_for_back;
                }
            }// end if (joyMask != joy)
            break;// end case save_file

        case wait_for_back:
            // Wait for the user to press the joystick before
            // returning to the main menu.
            joyMask = Joystick_GetStatus();
            if(joyMask != joy){
                joy = joyMask;
                if(joy & JOY_PRESS){
                    State = print_menu;
                }
            }
            break; // end case wait_for_back
    }// end switch(state)
}
```

Listing 1: app.c

A.2 DSP Functions

```c
#include "dsp_funcs.h"
#include "ring_buff.h"
#include "uda1380_mcb4300.h"
#include "arm_math.h"
#include "display_mcb4300.h"
#include <stdio.h>

// Maximum number of samples to use in the Welch estimate
#define WELCH_MAX RING_BUFF_SIZE

// Number of FIR filter coefficients and block size
#define NUM_TAPS 101
#define BLOCK_SIZE 2048

// Privcate variables
static uint32_t fftSize = FFT_POINTS;
static uint32_t ifftFlag = 0;
static uint32_t bitReverseFlag = 1;

// FIR filter coefficients (from Matlab)
const float32_t firCoeffs32[NUM_TAPS] = {
    5.03081370520049e-04f,5.09718215847710e-04f,5.23382468375693e-04f,
    5.43181681258115e-04f,5.67589845240078e-04f,5.94439943084953e-04f,
    6.20931471269369e-04f,6.43653243113658e-04f,6.58621471099649e-04f,
    6.61332803876887e-04f,6.46831671973846e-04f,6.09790982194298e-04f,
    5.44604900684948e-04f,4.45492185109446e-04f,3.06608273293462e-04f,
    1.22164114699374e-04f,-1.13450452885947e-04f,-4.05541121724168e-04f,
    -7.58983765706644e-04f,-1.17810181377248e-03f,-1.66654883522986e-03f,
    -2.22719877875602e-03f,-2.86204620916925e-03f,-3.57211874031211e-03f,
    -4.35740367090286e-03f,-5.21679059624895e-03f,-6.14803149645711e-03f,
    -7.14771949617298e-03f,-8.21128715764269e-03f,-9.33302481429553e-03f,
    -1.05061190828937e-02f,-1.17227113156839e-02f,-1.29739753772868e-02f,
    -1.42502137616257e-02f,-1.55409707093645e-02f,-1.68351606531324e-02f,
    -1.81212100129610e-02f,-1.93872100939972e-02f,-2.06210786082141e-02f,
    -2.18107271562588e-02f,-2.29442318686705e-02f,-2.40100043204098e-02f,
    -2.49969598009351e-02f,-2.58946800448127e-02f,-2.66935676049380e-02f,
    -2.73849891805852e-02f,-2.79614053934583e-02f,-2.84164847333577e-02f,
    -2.87451996668110e-02f,-2.89439032119143e-02f,9.70488023154390e-01f,
    -2.89439032119143e-02f,-2.87451996668110e-02f,-2.84164847333577e-02f,
    -2.79614053934583e-02f,-2.73849891805852e-02f,-2.66935676049380e-02f,
    -2.58946800448127e-02f,-2.49969598009351e-02f,-2.40100043204098e-02f,
    -2.29442318686705e-02f,-2.18107271562588e-02f,-2.06210786082141e-02f,
    -1.93872100939972e-02f,-1.81212100129610e-02f,-1.68351606531324e-02f,
    -1.55409707093645e-02f,-1.42502137616257e-02f,-1.29739753772868e-02f,
    -1.17227113156839e-02f,-1.05061190828937e-02f,-9.33302481429553e-03f,
    -8.21128715764269e-03f,-7.14771949617298e-03f,-6.14803149645711e-03f,
    -5.21679059624895e-03f,-4.35740367090286e-03f,-3.57211874031211e-03f,
    -2.86204620916925e-03f,-2.22719877875602e-03f,-1.66654883522986e-03f,
    -1.17810181377248e-03f,-7.58983765706644e-04f,-4.05541121724168e-04f,
    -1.13450452885947e-04f,1.22164114699374e-04f,3.06608273293462e-04f,
    4.45492185109446e-04f,5.44604900684948e-04f,6.09790982194298e-04f,
    6.46831671973846e-04f,6.61332803876887e-04f,6.58621471099649e-04f,
    6.43653243113658e-04f,6.20931471269369e-04f,5.94439943084953e-04f,
    5.67589845240078e-04f,5.43181681258115e-04f,5.23382468375693e-04f,
    5.09718215847710e-04f,5.03081370520049e-04f
};

// FIR state buffer
static float32_t firStateF32[BLOCK_SIZE + NUM_TAPS - 1];

// ARM FFT and FIR instances
arm_rfft_instance_f32 S;
arm_cfft_radix4_instance_f32 S_CFFT;
arm_fir_instance_f32 FS;

// Buffers used by the DSP functions
float32_t window[FFT_POINTS]__attribute__((at(FFT_WINDOW_ADDR)));
float32_t in[FFT_POINTS]__attribute__((at(FFT_IN_ADDR)));
float32_t out[FFT_POINTS*2]__attribute__((at(FFT_OUT_ADDR)));
float32_t mag[FFT_POINTS]__attribute__((at(FFT_MAG_ADDR)));
float32_t pxx[FFT_POINTS/2+1]__attribute__((at(PXX_BUFF_ADDR)));
```

```c
float32_t welch_buff[WELCH_MAX]__attribute__((at(WELCH_BUFF_ADDR)));
float32_t xfilt[WELCH_MAX]__attribute__((at(XFILT_ADDR)));

// External variables
extern volatile int rec_done;

/*---------------------------------------------------------------------------
  Initialize the DSP module, i.e. initialise FFT module and calculate the
  window to use for windowing the signal.
 *---------------------------------------------------------------------------*/
void dsp_init(void){
    static int i;
    uint32_t blockSize = BLOCK_SIZE;

    // Initialize the RFFT module
    arm_rfft_init_f32(&S, &S_CFFT, fftSize, ifftFlag, bitReverseFlag);

    // Initialize filter module
     arm_fir_init_f32(&FS, NUM_TAPS, (float32_t *)&firCoeffs32[0], &firStateF32[0],
          blockSize);

    // Initialize the window function
    for(i = 0; i < FFT_POINTS; i++){
        // Hamming window
        window[i] = 0.54 - 0.46 * cos ( 2 * PI * i /(FFT_POINTS-1));
    }
}

/*---------------------------------------------------------------------------
  Calculate the periodogram of the sampled signal
    - return: Pointer to the buffer that contains the periodogram values
 *---------------------------------------------------------------------------*/
float32_t * dsp_periodogram(void){
    static int i;

    __disable_irq(); // Start critical section
    ring_buff_set_read_index(FFT_POINTS);

    // Copy samples to buffer for FFT transform
    // and apply window function.
    for(i = 0; i < FFT_POINTS; i++){
        in[i] = (float32_t)ring_buff_read();
    }
    __enable_irq(); // End critical section

    // Process the data through the real FFT module
    arm_rfft_f32(&S, in, out);

    // Process the data through the Complex Magnitude Module to extraxt |X(f)|
    arm_cmplx_mag_f32(out, mag, fftSize/2+1);

    // Calculate one-sided Periodogram
    for(i = 0; i < FFT_POINTS/2+1; i++){
        // Correct for negative frequencies except 0 Hz and f_Nyquist
        if(i!=0 || i != FFT_POINTS/2){
            mag[i] *= 2;
        }
        //Pxx = (1/N)|X(f)|^2
        mag[i] = (1.0f/(float32_t)FFT_POINTS)*mag[i]*mag[i];
    }
    return mag;
}

/*---------------------------------------------------------------------------
  Calculate the Welch Estimate of the sampled signal
    - welch_length: Number of samples to use in the estimate
    - welch_overlap: Number of samples to overlap the periodograms
    - return: Pointer to the buffer that contains the estimate values
 *---------------------------------------------------------------------------*/
float32_t * dsp_welch(uint32_t welch_length, uint32_t welch_overlap){
    uint32_t blockSize = BLOCK_SIZE;
    uint32_t numBlocks = welch_length/BLOCK_SIZE;
    float32_t scale;
    static int i,j;
    int D = fftSize / (fftSize - welch_overlap); // Overlap fraction
    int K = welch_length/fftSize; // Number of estimates without overlap
    int L = D*(K-1)+1; // Total umber of estimates with overlap

    // Cakcykate scale to use for Pxx estimate
    scale = (1.0f/((float32_t)FFT_POINTS*(float32_t)L*0.3972f));

    __disable_irq(); // Start critical section
    ring_buff_set_read_index(welch_length);

    // Copy samples to buffer and scale down to -1 < x < 1.
    for(i = 0; i < welch_length; i++){
        welch_buff[i] = (float32_t)ring_buff_read()/2147483648.0f;
    }
    __enable_irq(); // End critical section

    // Clear Pxx buffer
    for(i = 0;i < FFT_POINTS/2+1;i++)
        pxx[i] = 0;

    // Filter the signal with FIR high-pass filter
     for(i=0; i < numBlocks; i++)
    {
      arm_fir_f32(&FS, welch_buff + (i * blockSize), xfilt + (i * blockSize), blockSize
           );
    }

    i = 0;
    // Non-circular overlapping segments
    while((i + FFT_POINTS - 1) < welch_length){

        // Perform windowing of current segment
        arm_mult_f32(xfilt+i,window,in,FFT_POINTS);

        // Process the data through the real FFT module
        arm_rfft_f32(&S, in, out);

        // Process the data through the Complex Magnitude Module to extraxt |X(f)|
        arm_cmplx_mag_f32(out, mag, fftSize/2+1);

        // Calculate one-sided power spectrum estimate
        for(j = 0; j < FFT_POINTS/2+1; j++){
            // Correct for negative frequencies except 0 Hz and f_Nyquist
            if(j != 0 || j != (FFT_POINTS/2)){
                mag[j] *= 2;
            }
            //Pxx = (1/MLU)|X(f)|^2
            pxx[j] += scale*(mag[j]*mag[j]);
        }
        // Update start index of next segment
        i += FFT_POINTS-welch_overlap;
    }
    //pxx[0] = pxx[1] = 0;
    return pxx;
}
```

Listing 2: dsp_funcs.c

A.3 Speed Measuring Algorithms

```
#include "speed.h"
#include "display_mcb4300.h"
#include "uda1380_mcb4300.h"
#include "sdmmc_mcb4300.h"
#include "dsp_funcs.h"
#include "arm_math.h"
#include "rtc.h"
#include "spectrum.h"
#include "board.h"
#include <stdio.h>

// Colors used by the speed module
#define SPEED_TEXT_COLOR LightGrey
#define SPEED_BACK_COLOR Black
#define SPEED_HEAD_COLOR Red

// Radar output frequency (24.100GHz)
#define F_OUT 24.1e9f
// Wave propagation speed
#define V_C 3e8f

// Parameters for Welch estimate
#define WELCH_N 20480
#define WELCH_OVERLAP 1536
#define WELCH_M FFT_POINTS

// Number of spectra for coherence estimate
#define NR_OF_SPEEDS 500
// Speed of the highest spectrum for coherence estimate
#define MAX_SPEED 250

// Filename whith spectra for coherence estimates
static char fname_s[] = "pyy.bin";

// Strings
static const char heading_s[] = "Speed Measurement";
static const char error_s[] = "ERROR:\nSPECTRUM FILE NOT FOUND!";
static const char calib_s[] = "Calibrating sensor. Please wait...";
static const char load_s[] = "Loading...";

// Local variables
static float32_t cos_th;
static int first = 1;
static char fname [16];
static uint8_t type;
static speed_alg_t algorithm = SPEED_WELCH;
static uint32_t npyy;
static uint32_t num_bytes;
static uint32_t timestamp[4];
static float32_t * v_vals;
static float32_t mean, var;
static RTC rtctime;

// Buffers used by speed algorithms
float32_t pyy_vals[NR_OF_SPEEDS*(WELCH_M/2+1)]__attribute__((at(PYY_START_ADDR)));
float32_t pxy[WELCH_M/2+1]__attribute__((at(PXY_START_ADDR)));
float32_t c_vals[NR_OF_SPEEDS];

// Shared variables
int speed_on = 0;

// External variables
extern volatile float maxval;
extern int adc_on;

/*---------------------------------------------------------------------------
  Calculate speed in km/h from Doppler frequency
    - f: Doppler Frequency
    - return: Speed in km/h
 *---------------------------------------------------------------------------*/
static float32_t speed_from_f(float32_t f){
```

```c
    return (f*V_C)/(2*F_OUT*cos_th)*3.6f;
}

/*-----------------------------------------------------------------------------
  Save a timestamp of the current time to the timestamp[] array
    - n: If 1, save start time. If 2, save end time.
 *----------------------------------------------------------------------------*/
static void do_timestamp(uint8_t n){
    // Get the current time
    rtc_gettime(&rtctime);

    if(n == 1){
        timestamp[0] = type << 24 | rtctime.hour << 16 | rtctime.min << 8 | rtctime.sec
              ;
        timestamp[1] = rtctime.year << 16 | rtctime.month << 8 | rtctime.mday;
    }
    else if (n == 2){
        timestamp[2] = rtctime.hour << 16 | rtctime.min << 8 | rtctime.sec;
        timestamp[3] = rtctime.year << 16 | rtctime.month << 8 | rtctime.mday;
    }
}

volatile int timer = 0;

/*-----------------------------------------------------------------------------
  Calibrate the speed module
 *----------------------------------------------------------------------------*/
void speed_calibrate(void){
    float32_t *pxx;
    float32_t mean1;
    float32_t var1;
    uint32_t i;

    display_print_string(0,0,calib_s);
    // Start the codec
    uda1380_start_stream();

    // Wait 1 second to get the number of samples needed to do one estimate.
    timer = 200;
    while(timer);

    mean = 0;
    var = 0;

    for(i=0;i<10;i++){
        // Get a welch estimate
        pxx = dsp_welch(WELCH_N,WELCH_OVERLAP);

        // Calculate mean value and standard deviation
        arm_mean_f32(pxx,(WELCH_M/2+1),&mean1);
        arm_var_f32(pxx,(WELCH_M/2+1),&var1);

        mean += mean1;
        var += var1;
    }

    mean /= 10.0f;
    var /= 10.0f;

    // Stop the codec
    uda1380_stop_stream();
}


/*-----------------------------------------------------------------------------
  Initialize the speed module
    - angle: The radar tilt angle in radians
    - alg: The algorithm type to use: SPEED_PERIODOGRAM, SPEED_WELCH or
           SPEED_COHERENCE
 *----------------------------------------------------------------------------*/
void speed_init(float32_t angle, speed_alg_t alg){
    static ADC_Clock_Setup_Type ADCSetup;

    // Set estimation time and load spectra from file if
    // coherence estimate is selected.
    switch(alg){
        case SPEED_PERIODOGRAM:
                type = SPEED_PERIODOGRAM;
            break;
        case SPEED_WELCH:
                type = SPEED_WELCH;
            break;
        case SPEED_COHERENCE:
            if(sdmmc_read_dir() == -1){
                display_print_string(1,0,error_s);
                return;
            }
            display_print_string(2,0,load_s);
            sdmmc_load_float(fname_s, pyy_vals, &npyy);
            if(npyy == 0){
                display_print_string(1,0,error_s);
                return;
            }
            type = SPEED_COHERENCE;
            break;
    }

    // Set up display
    display_clear(SPEED_BACK_COLOR);
    display_set_font_size(BIG_FONT);
    display_set_back_color(SPEED_BACK_COLOR);
    display_set_text_color(SPEED_HEAD_COLOR);
    display_print_string(0,0,heading_s);
    display_set_text_color(SPEED_TEXT_COLOR);
    first = 1;

    // Calculate cos(theta)
    cos_th = cos(angle);

    // Save algorithm type and initialize pointer to where to save
    // the estimates
    algorithm = alg;
    v_vals  = (float32_t *)REC_START_ADDR;
    num_bytes = 0;

    // Save start timestamp
    do_timestamp(1);

    // Save filename - on form DDHHMMSS.DOP
    snprintf(fname,16,"%02d%02d%02d%02d.DOP",
        rtctime.mday,rtctime.hour,rtctime.min,rtctime.sec);

    // ENABLE ADC TO CONTROL MAX LEVEL SENSITIVITY
    Board_ADC_Init();
    Chip_ADC_Init(LPC_ADC0, &ADCSetup);
    Chip_ADC_Channel_Enable_Cmd(LPC_ADC0, ADC_CH1, ENABLE);
    Chip_ADC_Set_Resolution(LPC_ADC0, &ADCSetup, ADC_10BITS);
    Chip_ADC_Set_SampleRate(LPC_ADC0, &ADCSetup, 400000);
    // Enable ADC Interrupt
    NVIC_ClearPendingIRQ(ADC0_IRQn);
    NVIC_EnableIRQ(ADC0_IRQn);
    Chip_ADC_Channel_Int_Cmd(LPC_ADC0, ADC_CH1, ENABLE);

    // Start the codec and set flag for SysTick handler
    uda1380_start_stream();
    adc_on = 1;
    speed_on = 1;
}

/*----------------------------------------------------------------------------
  De-Initialize the speed module. Saves the estimate to file on SD card.
 *----------------------------------------------------------------------------*/
void speed_deinit(void){
    // Stop codec and clear flag for SysTick handler
    uda1380_stop_stream();
    speed_on = 0;

    // Save timestamp
    do_timestamp(2);

    // Save measurements to file
    sdmmc_save_speed(fname,timestamp,(float32_t *)REC_START_ADDR,num_bytes);

    // Stop the ADC
    NVIC_DisableIRQ(ADC0_IRQn);
    Chip_ADC_Channel_Int_Cmd(LPC_ADC0, ADC_CH1, DISABLE);
    Chip_ADC_Channel_Enable_Cmd(LPC_ADC0, ADC_CH1, DISABLE);
    Chip_ADC_DeInit(LPC_ADC0);

    // Clear flags for SysTick handler
    adc_on = 0;
}

/*----------------------------------------------------------------------------
  The main function for speed calculation
 *----------------------------------------------------------------------------*/
void speed(void){
    char b[128];
    float32_t *pxx;
    float32_t max_val, mean_val;
    float32_t dv, df;

    uint32_t i,j,k;
    static float32_t dv_old = 0;

    // Do estimation according to selected algorithm
    switch(algorithm){
    case SPEED_PERIODOGRAM:
        // Get one single periodogram
        pxx = dsp_periodogram();
        // Calculates max value and returns corresponding value and index
        arm_max_f32(pxx, FFT_POINTS/2+1, &max_val, &i);
        // Calculate the corresponding frequency
        df = (((float32_t)i/(float32_t)FFT_POINTS)*(float32_t)SAMPLE_RATE);
        // Calculate relative speed in km/h
        dv = speed_from_f(df);
        break; // end case SPEED_PERIODOGRAM

    case SPEED_WELCH:
        // Get the welch estimate
        pxx = dsp_welch(WELCH_N,WELCH_OVERLAP);

        // Get max value
        arm_max_f32(pxx, WELCH_M/2+1, &max_val, &i);
        // Calculate the corresponding frequency
        df = (((float32_t)i/(float32_t)WELCH_M)*(float32_t)SAMPLE_RATE);
        // Calculate relative speed in km/h
        dv = speed_from_f(df);

        // Check if max value is large enough
        arm_mean_f32(pxx, WELCH_M/2+1, &mean_val);
        if (mean_val < (mean * 3.0f * maxval)){
            dv = 0;
            df = 0;
            i = 0;
        }
        break; // end case SPEED_WELCH

    case SPEED_CORRELATION:
        // Clear coherence value array
        for(j = 0; j < NR_OF_SPEEDS; j++){
            c_vals[j] = 0;
        }
        // Get the Welch estimate
        pxx = dsp_welch(WELCH_N,WELCH_OVERLAP);

        // Compare coherence with pre-estimated spectra
        for(j = 0; j < NR_OF_SPEEDS; j++){
            // PXY = PXX * PYY
            arm_mult_f32(pxx,&pyy_vals[j*(WELCH_M/2+1)],pxy,WELCH_M/2+1);
            // sum(pxy)
            for(k = 0; k < WELCH_M/2+1; k++){
                c_vals[j] += pxy[k];
            }
        }
        // Calculates max coherence value and returns corresponding value and index
        arm_max_f32(c_vals, NR_OF_SPEEDS, &max_val, &i);
        // Calculate the corresponding speed
        dv = ((float32_t)i/((float32_t)NR_OF_SPEEDS/(float32_t)MAX_SPEED));

        // Check if max value is large enough
        arm_mean_f32(pxx, WELCH_M/2+1, &mean_val);
        if (mean_val < (mean * 3.0f * maxval)){
            dv = 0;
            df = 0;
            i = 0;
        }
        break;// end case SPEED_COHERENCE
    }// End switch(alg)

    // If the speed has changed, update the display
    if(dv != dv_old || first){
        if(first)
            first = 0;

        dv_old = dv;
        // Logic to make shure that sudden, unrealistic increases or decreases of
        // speed is not recorded. Speed is set to previous value in stead.
/*      if((dv > dv_old + 20.0f) || (dv < dv_old - 20.0f))
            dv = dv_old;
        else
            dv_old = dv;
*/
        // Print result on screen
        snprintf(b,128,"Max @ %04d/%05.0fHz\n\nSpeed: %05.1f m/s\n       %05.1f km/h",
            i,df,dv/3.6f,dv);
        display_print_string(2,0,b);
    }

    // Record values until out of memory
    if(v_vals < (float32_t *)REC_END_ADDR){
        *v_vals = dv;
        v_vals ++;
        num_bytes += 4;
    }
}// end speed()
```

Listing 3: speed.c

A.4 CODEC Driver

```c
#include "uda1380_mcb4300.h"
#include "board.h"
#include "ring_buff.h"
#include "sdmmc_mcb4300.h"

// UDA1380 Register variables
// System Register Data Set
uint16_t UDA1380_sys_regs_dat[] = {
    UDA1380_REG_EVALCLK_VAL,
    UDA1380_REG_I2S_VAL,
    UDA1380_REG_PWRCTRL_VAL,
    UDA1380_REG_ANAMIX_VAL,
    UDA1380_REG_HEADAMP_VAL
};

// Interpolator Register Data Set
uint16_t UDA1380_interfil_regs_dat[] = {
    UDA1380_REG_MSTRVOL_VAL,
    UDA1380_REG_MIXVOL_VAL,
    UDA1380_REG_MODEBBT_VAL,
    UDA1380_REG_MSTRMUTE_VAL,
    UDA1380_REG_MIXSDO_VAL
};
// Decimator Register Data Set
uint16_t UDA1380_decimator_regs_dat[] = {
    UDA1380_REG_DECVOL_VAL,
    UDA1380_REG_PGA_VAL,
    UDA1380_REG_ADC_VAL,
    UDA1380_REG_AGC_VAL
};

// Pointer to memory for writing
static volatile int32_t *wrptr = (int32_t *)REC_START_ADDR;

// Private variables
static int uda1380_mode = 0;
static uint32_t rec_samples;
static volatile uint32_t br;
static int32_t rec_was_stopped = 1;

// Shared variables
volatile int rec_done = 0;
volatile int play_done = 0;

/*---------------------------------------------------------------------------
  IRQ Handler for the I2S Peripheral
 *---------------------------------------------------------------------------*/
void I2S0_IRQHandler(void)
{
    if(uda1380_mode == UDA1380REC){
        // Copy the samples from the I2S RX buffer to the wrptr destination.
        // When done, stop recording and set flag.
        while ((Chip_I2S_GetLevel(LPC_I2S0, I2S_RX_MODE) > 0)) {
            if (wrptr <= (int32_t *)REC_START_ADDR + rec_samples-1){
                *wrptr = Chip_I2S_Receive(LPC_I2S0);
                wrptr++;
                br+=4;
            }
            else{
                uda1380_stop_rec();
                rec_done = 1;
                break;
            }
        }
    }
    else if(uda1380_mode == UDA1380STREAM){
        // Copy samples from the I2S RX buffer to the ring buffer
        while (Chip_I2S_GetLevel(LPC_I2S0, I2S_RX_MODE) > 0) {
            ring_buff_write(Chip_I2S_Receive(LPC_I2S0));
        }
    }
```

```
}


/*----------------------------------------------------------------------------
  Very simple (inaccurate) delay function
    - i: delay time in cycles
 *----------------------------------------------------------------------------*/
static void delay(uint32_t i) {
    while (i--) {}
}

/*----------------------------------------------------------------------------
  Write data to UDA registers
    - reg: Address of the register
    - value: Value to write
    - I2C_Config: Pointer to the I2C config structure
 *----------------------------------------------------------------------------*/
static void UDA_Reg_write(UDA1380_REG_t reg, unsigned short value, I2C_M_SETUP_Type *
      I2C_Config) {
    I2C_Config->tx_data[0] = reg;
    I2C_Config->tx_data[1] = value >> 8;
    I2C_Config->tx_data[2] = value & 0xFF;
    Chip_I2C_MasterTransmitData(LPC_I2C0, I2C_Config, I2C_TRANSFER_POLLING);
    delay(10000);
}

/*----------------------------------------------------------------------------
  Read data from UDA registers
    - reg: Address of the register
    - return: Value of the register
 *----------------------------------------------------------------------------*/
static uint16_t UDA_Reg_read(UDA1380_REG_t reg) {
    uint8_t rx_data[2];
    Chip_I2C_MasterReadReg(LPC_I2C0, UDA1380_I2C_ADDR, reg, rx_data, 2);
    return rx_data[0] << 8 | rx_data[1];
}

/*----------------------------------------------------------------------------
  Initialize UDA1380 codec
    - audio_in_sel: UDA1380_AUDIO_LINE_IN_SELECT for line in or
                    UDA1380_AUDIO_MIC_SELECT for mic
    - return: 0 if successful, -1 otherwise
 *----------------------------------------------------------------------------*/
int uda1380_init(UDA1380_Input_Sel_t audio_in_sel){
    uint16_t temp;
    uint8_t  i;
    uint8_t uda1380_tx_data_buf[3];

    Chip_I2S_Audio_Format_Type I2S_Config;
    I2C_M_SETUP_Type I2C_Config;
    // I2S Configuration
    I2S_Config.SampleRate = SAMPLE_RATE;
    I2S_Config.ChannelNumber = CHANNELS;
    I2S_Config.WordWidth =  WORDWIDTH;
    // I2C Configuration
    I2C_Config.sl_addr7bit = I2CDEV_UDA1380_ADDR;
    I2C_Config.retransmissions_max = 5;
    I2C_Config.tx_length = 3;
    I2C_Config.tx_data = uda1380_tx_data_buf;
    I2C_Config.rx_length = 0;
    I2C_Config.rx_data = NULL;

    // Initialize I2C peripheral
    Chip_I2C_Init(LPC_I2C0);
    Chip_I2C_SetClockRate(LPC_I2C0, 400000);
    Chip_I2C_Cmd(LPC_I2C0, I2C_MASTER_MODE, ENABLE);

    // Start I2S Peripheral to supply clock to UDA1380 before reset
    Chip_I2S_Init(LPC_I2S0);
    Chip_I2S_Config(LPC_I2S0, I2S_RX_MODE, &I2S_Config);
    Chip_I2S_Config(LPC_I2S0, I2S_TX_MODE, &I2S_Config);

    // Initialize UDA1380 CODEC
    // Reset UDA1380 on board
    Chip_SCU_PinMux(0x8, 0, MD_PUP, FUNC0);
    Chip_GPIO_WriteDirBit(4, 0, true);
    Chip_GPIO_WritePortBit(4, 0, true);
    // delay 1us
    delay(100000);
    Chip_GPIO_WritePortBit(4, 0, false);
    delay(100000);

    // Write startup values to UDA13800 registers
    // System regs
    for (i = 0; i < 5; i++) {
        UDA_Reg_write((UDA1380_REG_t) (UDA_REG_EVALM_CLK + i), UDA1380_sys_regs_dat[i],
              &I2C_Config);
        temp = UDA_Reg_read((UDA1380_REG_t) (UDA_REG_EVALM_CLK + i));
        if (temp != UDA1380_sys_regs_dat[i]) {
            return -1;
        }
    }

    // Interpolator regs
    for (i = 0; i < 5; i++) {
        UDA_Reg_write((UDA1380_REG_t) (UDA_REG_MASTER_VOL_CTRL + i),
             UDA1380_interfil_regs_dat[i], &I2C_Config);
        temp = UDA_Reg_read((UDA1380_REG_t) (UDA_REG_MASTER_VOL_CTRL + i));
        if (temp != UDA1380_interfil_regs_dat[i]) {
            return -1;
        }
    }
    // Decimator regs
    for (i = 0; i < 4; i++) {
        UDA_Reg_write((UDA1380_REG_t) (UDA_REG_DEC_VOL_CTRL + i),
             UDA1380_decimator_regs_dat[i], &I2C_Config);
        temp = UDA_Reg_read((UDA1380_REG_t) (UDA_REG_DEC_VOL_CTRL + i));
        if (temp != UDA1380_decimator_regs_dat[i]) {
            return -1;
        }
    }

    // If mic is selected as the input
    if (audio_in_sel == UDA1380_AUDIO_MIC_SELECT) {
        // Disable Power On for ADCR, PGAR, PGAL
        UDA_Reg_write((UDA1380_REG_t) (UDA_REG_POWER_CTRL), UDA1380_REG_PWRCTRL_VAL &
             (~(0x000B)), &I2C_Config);
        temp = UDA_Reg_read((UDA1380_REG_t) (UDA_REG_POWER_CTRL));
        if (temp != (UDA1380_REG_PWRCTRL_VAL & (~(0x000B)))) {
            return -1;
        }
        // Enable Microphone input
        UDA_Reg_write((UDA1380_REG_t) (UDA_REG_ADC_CTRL), UDA1380_REG_ADC_VAL |
             UDA1380_AUDIO_MIC_SELECT, &I2C_Config);
        temp = UDA_Reg_read((UDA1380_REG_t) (UDA_REG_ADC_CTRL));
        if (temp != (UDA1380_REG_ADC_VAL | UDA1380_AUDIO_MIC_SELECT)) {
            return -1;
        }
    }// End if UDA1380_AUDIO_MIC_SELECT
    return 0;
}

/*---------------------------------------------------------------------------
  Start recording
    - samples: Number of samples to reord
    - dst: Pointer to destination address
 *---------------------------------------------------------------------------*/
void uda1380_start_rec(uint32_t samples, int32_t * dst){
    rec_done = 0;
    br = 0;
    rec_samples = samples;
    uda1380_mode = UDA1380REC;
    wrptr = dst;
    Chip_I2S_Int_Cmd(LPC_I2S0, I2S_RX_MODE,    ENABLE, 4);
    Chip_I2S_Start(LPC_I2S0, I2S_RX_MODE);
    // Enable interrupts
    NVIC_ClearPendingIRQ(I2S0_IRQn);
    NVIC_EnableIRQ(I2S0_IRQn);
```

```
}

/*---------------------------------------------------------------------------
  Pause recording
 *---------------------------------------------------------------------------*/
void uda1380_pause_rec(void){
    NVIC_DisableIRQ(I2S0_IRQn);
    Chip_I2S_Pause(LPC_I2S0, I2S_RX_MODE);
    rec_was_stopped = 0;
}

/*---------------------------------------------------------------------------
  Resume recording (if paused)
 *---------------------------------------------------------------------------*/
void uda1380_resume_rec(void){
    if(rec_was_stopped)
        return;
    else{
        rec_done = 0;
        uda1380_mode = UDA1380REC;
        Chip_I2S_Start(LPC_I2S0, I2S_RX_MODE);
        // Enable interrupts
        NVIC_ClearPendingIRQ(I2S0_IRQn);
        NVIC_EnableIRQ(I2S0_IRQn);
    }
}

/*---------------------------------------------------------------------------
  Stop Recording
    - return: The number of BYTES recorded
 *---------------------------------------------------------------------------*/
uint32_t uda1380_stop_rec(void){
    NVIC_DisableIRQ(I2S0_IRQn);
    Chip_I2S_Int_Cmd(LPC_I2S0, I2S_RX_MODE,    DISABLE, 4);
    Chip_I2S_Stop(LPC_I2S0, I2S_RX_MODE);
    rec_was_stopped = 1;
    return br;
}

/*---------------------------------------------------------------------------
  Start Streaming
 *---------------------------------------------------------------------------*/
void uda1380_start_stream(void){
    // Start I2S Audio Stream
    Chip_I2S_Start(LPC_I2S0, I2S_RX_MODE);

    // Enable interrupts
    uda1380_mode = UDA1380STREAM;
    Chip_I2S_Int_Cmd(LPC_I2S0, I2S_RX_MODE,    ENABLE, 4);
    NVIC_ClearPendingIRQ(I2S0_IRQn);
    NVIC_EnableIRQ(I2S0_IRQn);
}

/*---------------------------------------------------------------------------
  Stop Streaming
 *---------------------------------------------------------------------------*/
void uda1380_stop_stream(void){
    NVIC_DisableIRQ(I2S0_IRQn);

    // Disable interrupts
    Chip_I2S_Int_Cmd(LPC_I2S0, I2S_RX_MODE,    DISABLE, 4);

    // Stop I2S Audio Stream
    Chip_I2S_Stop(LPC_I2S0, I2S_RX_MODE);
}
```

Listing 4: uda1380_mcb4300.c

A.5 SD Card Interface Drives

```
#include "sdmmc_mcb4300.h"
#include "uda1380_mcb4300.h"
#include "rtc.h"
#include "ff.h"
#include "board.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Buffer size (in bytes) for R/W operations
#define BUFFER_SIZE     4096

// File system object
static FATFS Fatfs;

// Variables
static volatile UINT Timer = 0;
static volatile int32_t sdio_wait_exit = 0;

// SDMMC card info structure
mci_card_struct sdcardinfo;

/*---------------------------------------------------------------------------
  WAVE file header fields
 *---------------------------------------------------------------------------*/
static char ckID[] = "RIFF";                    // RIFF chunk
static int32_t cksize;                          // Chunk size
static char WAVEID[] = "WAVE";                  // WAVEID
static char ckID2[] = "fmt ";                   // Format Chunk
static int32_t cksize2 = 16;                    // Chunk size
static int16_t wFormatTag = 1;                  // WAVE_FORMAT_EXTENSIBLE
static int16_t nChannels = 1;                   // Channels = 1 (mono)
static int32_t nSamplesPerSec = SAMPLE_RATE;    // Sampling Freq.
static int32_t nAvgBytesPerSec = SAMPLE_RATE*4; // Bytes Pr sec (Fs*4)
static int16_t nBlockAlign = 4;                 // Bytes Pr sample (4 = 32 bits)
static int16_t wBitsPerSample = 32;             // Bits Pr sample (32)
static char ckID3[] = "data";                   // Data chunk
static int32_t cksize3;                         // Chunk size

/*---------------------------------------------------------------------------
  Simple wait function
    - time: Wait time in ms
 *---------------------------------------------------------------------------*/
static void sdmmc_waitms(uint32_t time){
    /* In an RTOS, the thread would sleep allowing other threads to run.
       For standalone operation, we just spin on RI timer */
    int32_t curr = (int32_t) Chip_RIT_GetCounter();
    int32_t final = curr + ((SystemCoreClock / 1000) * time);

    if (final == curr) return;

    if ((final < 0) && (curr > 0)) {
        while (Chip_RIT_GetCounter() < (uint32_t) final) {}
    }
    else {
        while ((int32_t) Chip_RIT_GetCounter() < final) {}
    }

    return;
}


/*---------------------------------------------------------------------------
  Sets up the SD event driven wakeup
    - bits : Status bits to poll for command completion
 *---------------------------------------------------------------------------*/
static void sdmmc_setup_wakeup(uint32_t bits){
    // Wait for IRQ - for an RTOS, you would pend on an event here with a IRQ based
         wakeup.
    NVIC_ClearPendingIRQ(SDIO_IRQn);
    sdio_wait_exit = 0;
    Chip_SDMMC_SetIntMask(bits);
    NVIC_EnableIRQ(SDIO_IRQn);
}


/*---------------------------------------------------------------------------
  A better wait callback for SDMMC driven by the IRQ flag
    - return: 0 on success, or failure condition (-1)
 *---------------------------------------------------------------------------*/
static uint32_t sdmmc_irq_driven_wait(void){
    uint32_t status;

    // Wait for event, would be nice to have a timeout, but keep it  simple
    while (sdio_wait_exit == 0) {}
    // Get status and clear interrupts
    status = Chip_SDMMC_GetIntStatus();

    return status;
}


/*---------------------------------------------------------------------------
  Print the result code
    - rc: result code
 *---------------------------------------------------------------------------*/
static void die(FRESULT rc){
    printf("Failed with rc=%u.\n", rc);
}

/*---------------------------------------------------------------------------
  SDIO controller interrupt handler
 *---------------------------------------------------------------------------*/
void SDIO_IRQHandler(void){
    /* All SD based register handling is done in the callback
       function. The SDIO interrupt is not enabled as part of this
       driver and needs to be enabled/disabled in the callbacks or
       application as needed. This is to allow flexibility with IRQ
       handling for applicaitons and RTOSes. */
    /* Set wait exit flag to tell wait function we are ready. In an RTOS,
       this would trigger wakeup of a thread waiting for the IRQ. */
    NVIC_DisableIRQ(SDIO_IRQn);
    sdio_wait_exit = 1;
}

/*---------------------------------------------------------------------------
  SDIO controller init routine
 *---------------------------------------------------------------------------*/
void sdmmc_init(void){

    // Initialize SD/MMC
    memset(&sdcardinfo, 0, sizeof(sdcardinfo));
    sdcardinfo.evsetup_cb = sdmmc_setup_wakeup;
    sdcardinfo.waitfunc_cb = sdmmc_irq_driven_wait;
    sdcardinfo.msdelay_func = sdmmc_waitms;

    Board_SDMMC_Init();
    Chip_SDMMC_Init();

    NVIC_DisableIRQ(SDIO_IRQn);
    NVIC_EnableIRQ(SDIO_IRQn);
    f_mount(0, &Fatfs);     // Register volume work area (never fails)
    NVIC_DisableIRQ(SDIO_IRQn);
}

/*---------------------------------------------------------------------------
  Save src to SD-card in WAV format
    - fname: Filename
    - src: Pointer to source address
    - num_bytes: Mumber of bytes to write to file
 *---------------------------------------------------------------------------*/
void sdmmc_save_wav(char * fname, int32_t * src, uint32_t num_bytes){
    static FRESULT rc;
    static FIL fd;
    static unsigned int bw;
```

```c
    char *wbuf = (char * ) src;

    // Calculate WAV chunck sizes
    cksize = 4+24+8+num_bytes;      // 4 + 48 + 12 + (8 + M * Nc * Ns)
    cksize3 = num_bytes;            // M * Nc * Ns

    NVIC_EnableIRQ(SDIO_IRQn);

    // Open the file
    rc = f_open(&fd, fname, FA_WRITE | FA_CREATE_ALWAYS);
    if (rc) die(rc);

    // Write WAV file header
    rc = f_write(&fd, ckID, 4, &bw);
    if (rc) die(rc);
    rc = f_write(&fd, (char * )&cksize, 4, &bw);
    if (rc) die(rc);
    rc = f_write(&fd, WAVEID, 4, &bw);
    if (rc) die(rc);
    rc = f_write(&fd, ckID2, 4, &bw);
    if (rc) die(rc);
    rc = f_write(&fd, (char * )&cksize2, 4, &bw);
    if (rc) die(rc);
    rc = f_write(&fd, (char * )&wFormatTag, 2, &bw);
    if (rc) die(rc);
    rc = f_write(&fd, (char * )&nChannels, 2, &bw);
    if (rc) die(rc);
    rc = f_write(&fd, (char * )&nSamplesPerSec, 4, &bw);
    if (rc) die(rc);
    rc = f_write(&fd, (char * )&nAvgBytesPerSec, 4, &bw);
    if (rc) die(rc);
    rc = f_write(&fd, (char * )&nBlockAlign, 2, &bw);
    if (rc) die(rc);
    rc = f_write(&fd, (char * )&wBitsPerSample, 2, &bw);
    if (rc) die(rc);
    rc = f_write(&fd, ckID3, 4, &bw);
    if (rc) die(rc);
    rc = f_write(&fd, (char * )&cksize3, 4, &bw);
    if (rc) die(rc);

    // Write data to file
    rc = f_write(&fd, wbuf, num_bytes, &bw);
    if (rc) die(rc);

    // Close the file
    rc = f_close(&fd);
    if (rc) die(rc);

    NVIC_DisableIRQ(SDIO_IRQn);
}

/*---------------------------------------------------------------------------
  Load file with floating-point values from SD-card to dst
    - fname: Filename
    - dst: Floating-point pointer to destination address
    - bytes_read: Will contain the number of bytes read when done
 *---------------------------------------------------------------------------*/
void sdmmc_load_float(char * fname, float32_t * dst, uint32_t * bytes_read){
    static int32_t buffer[BUFFER_SIZE/sizeof(int32_t)];
    static FRESULT rc;  /* Result code */
    static FIL fd;      /* File object */
    static unsigned int br, i;
    char *cbuf = (char * ) buffer;
    char *rbuf = (char * ) dst;
    uint32_t bufi = 0;

    NVIC_EnableIRQ(SDIO_IRQn);

    // Open the file
    rc = f_open(&fd, fname, FA_READ);
    if (rc) die(rc);

    for (;; ) {
        // Read a chunk of file
        rc = f_read(&fd, (void *)cbuf, BUFFER_SIZE, &br);
        if (rc || !br) {
            break;                  // Error or end of file
        }
        for (i = 0; i < br; i++){   // Copy data to destination
            rbuf[bufi] = cbuf[i];
            bufi++;
        }
    }
    if (rc) die(rc);

    *bytes_read = bufi;

    // Close the file
    rc = f_close(&fd);
    if (rc) die(rc);

    NVIC_DisableIRQ(SDIO_IRQn);
}

/*-----------------------------------------------------------------------------
  Save speed measurements to SD-card. First 16 bytes are timestamp info.
    - fname: Filename
    - timestamp: Pointer to the 16-byte timestamp array
    - src: Pointer to source address
    - num_bytes: Mumber of bytes to write to file
 *-----------------------------------------------------------------------------*/
void sdmmc_save_speed(char * fname, uint32_t * timestamp, float32_t * src, uint32_t
      num_bytes){
    static FRESULT rc;  // Result code
    static FIL fd;      // File object
    static unsigned int bw;
    char *tbuf = (char * ) timestamp;
    char *wbuf = (char * ) src;

    NVIC_EnableIRQ(SDIO_IRQn);

    // Open the file
    rc = f_open(&fd, fname, FA_WRITE | FA_CREATE_ALWAYS);
    if (rc) die(rc);

    // Write timestamp
    rc = f_write(&fd, tbuf, 16, &bw);
    if (rc) die(rc);

    // Write data
    rc = f_write(&fd, wbuf, num_bytes, &bw);
    if (rc) die(rc);

    // Close the file
    rc = f_close(&fd);
    if (rc) die(rc);

    NVIC_DisableIRQ(SDIO_IRQn);
}

/*-----------------------------------------------------------------------------
  Read the root directory listing
    - return: Number of files in directory
 *-----------------------------------------------------------------------------*/
int32_t sdmmc_read_dir(void){
    static FRESULT rc;
    static DIR dir;
    static FILINFO fno;
    static int i;

    i = 0;
    NVIC_EnableIRQ(SDIO_IRQn);

    rc = f_opendir(&dir, "");
    if (rc) die(rc);

    //printf("\f");
    for (;; ) {
        // Read a directory item
        rc = f_readdir(&dir, &fno);
        if (rc || !fno.fname[0]) {
            break;                  // Error or end of dir
        }
        if (fno.fattrib & AM_DIR) {
            //printf("   <dir>  %s\r\n", fno.fname);
        }
        else {
            //printf("\n   %s", fno.fname);
            i++;
        }
    }
    if (rc) die(rc);

    NVIC_DisableIRQ(SDIO_IRQn);
    if(i == 0)
        i=-1;
    return i;
}
```

Listing 5: sdmmc_mcb4300.c

A.6 Display Driver

```c
#include "display_mcb4300.h"
#include "memory_map.h"
#include "board.h"
#include "Font_6x8_h.h"
#include "Font_16x24_h.h"

#define PHYS_XSZ    240                /* Physical screen width              */
#define PHYS_YSZ    320                /* Physical screen height             */
#define DELAY_2N    18                 /* Increase delay @ high freqs        */
#define BG_COLOR    0                  /* Background color                   */
#define TXT_COLOR   1                  /* Text color                         */

#define BIG_FONT_HEIGHT 24
#define BIG_FONT_WIDTH  16
#define SMALL_FONT_HEIGHT 8
#define SMALL_FONT_WIDTH 6

// pointer to frame buffer
static uint16_t framebuffer[PHYS_XSZ*PHYS_YSZ]__attribute__((at(FRAME_BUFFER_ADDR)));

// Private variables
static uint16_t Color[2] = {White, Black};
static uint8_t landscape = 0;
static uint16_t width = PHYS_XSZ;
static uint16_t height = PHYS_YSZ;
static uint16_t font_size = 0;
static uint16_t font_width = SMALL_FONT_WIDTH;
static uint16_t font_height = SMALL_FONT_HEIGHT;

/*-------------------------------------------------------------------------
  Very simple (inaccurate) delay function. DELAY_2N will increase the delay
  time by 2^N.
    - cnt: delay time
 *-------------------------------------------------------------------------*/
static void delay (int cnt) {
  cnt <<= DELAY_2N;
  while (cnt--);
}

/*-------------------------------------------------------------------------
  Draw character from font at x,y.
    - x: X position
    - y: Y position
    - cw: Character width
    - ch: Character height
    - c: pointer to the character in the font file
 *-------------------------------------------------------------------------*/
static void DrawChar (unsigned int x, unsigned int y, unsigned int cw, unsigned int ch,
      char *c) {
  unsigned int i, j, k, pixs;

  k  = (cw + 7)/8; // bytes pr char

  if (k == 1) {
    for (j = 0; j < ch; j++) {
      pixs = *(char *)c;
      c += 1;
      for (i = 0; i < cw; i++) {
        if (landscape == 0)
            framebuffer[(y+j)*PHYS_XSZ + (x+i)] = Color[(pixs >> i) & 1];
        else
            framebuffer[(y+j) + ((PHYS_YSZ-1)-x-i)*PHYS_XSZ] = Color[(pixs >> i) & 1];
      }
    }
  }
  else if (k == 2) {
    for (j = 0; j < ch; j++) {
      pixs = *(unsigned short *)c;
      c += 2;

      for (i = 0; i < cw; i++) {
        if (landscape == 0)
            framebuffer[(y+j)*PHYS_XSZ + (x+i)] = Color[(pixs >> i) & 1];
        else
            framebuffer[(y+j) + ((PHYS_YSZ-1)-x-i)*PHYS_XSZ] = Color[(pixs >> i) & 1];
      }
    }
  }
}

/*-----------------------------------------------------------------------------
  Scroll the display from the bottom up
    - dy: Scroll-distance in pixels
 *----------------------------------------------------------------------------*/
static void ScrollVertical (unsigned int dy) {
    if (landscape == 0){
      uint32_t x, y;
      for (y = 0; y < (PHYS_YSZ - dy); y++) {
        for (x = 0; x < PHYS_XSZ; x++) {
          framebuffer[y*PHYS_XSZ + x] = framebuffer[(y+dy)*PHYS_XSZ + x];
        }
      }
      for (; y < PHYS_YSZ; y++) {
        for (x = 0; x < PHYS_XSZ; x++) {
          framebuffer[y*PHYS_XSZ + x] = Color[BG_COLOR];
        }
      }
    }
}


/*-----------------------------------------------------------------------------
  Clear the display
    - color: Color
 *----------------------------------------------------------------------------*/
void display_clear(unsigned short color) {
  unsigned int i;
  for (i = 0; i < (PHYS_XSZ*PHYS_YSZ); i++) {
    framebuffer[i] = color;
  }
}

/*-----------------------------------------------------------------------------
  Put one pixel at x,y
    - x: X position
    - y: Y position
    - color: Color
 *----------------------------------------------------------------------------*/
void display_put_pixel (unsigned int x, unsigned int y, uint16_t color) {
    if (landscape == 0)
        framebuffer[y*PHYS_XSZ + x] = color;
    else
        framebuffer[x*PHYS_XSZ + y] = color;
}

/*-----------------------------------------------------------------------------
  Draw a line from one point
    - x: X start position
    - y: Y start position
    - len: Lenght in pixels
    - dir: Direction - 0: Vertical, 1: Horizontal
    - color: Color
 *----------------------------------------------------------------------------*/
void display_draw_line (unsigned int x, unsigned int y, unsigned int len, unsigned int
     dir, uint16_t color) {
  uint32_t i;

  if (dir == 0) {                       /* Vertical line                      */
    for (i = 0; i < len; i++) { display_put_pixel (x, y - i, color); }
  }
  else {                                /* Horizontal line                    */
    for (i = 0; i < len; i++) { display_put_pixel (x + i, y, color); }
  }
}

```

```c
/*---------------------------------------------------------------------------
  Print a character at line, col
    - ln: Line number
    - col: Column number
    - c: Character to print
 *---------------------------------------------------------------------------*/
void display_print_char (unsigned int ln, unsigned int col, char c) {
    c -= 32;
    switch (font_size) {
        case SMALL_FONT:  /* Font 6 x 8 */
            DrawChar(col *  6, ln *  8,  6,  8, (char *)&Font_6x8_h  [c * 8]);
            break;
        case BIG_FONT:  /* Font 16 x 24 */
            DrawChar(col * 16, ln * 24, 16, 24, (char *)&Font_16x24_h[c * 24]);
            break;
    }
 }

/*---------------------------------------------------------------------------
  Print a string at line, col
    - ln: Line number
    - col: Column number
    - c: Pointer to the string to print
 *---------------------------------------------------------------------------*/
void display_print_string (unsigned int ln, unsigned int col, const char *s) {
    while (*s) {
        if(*s == '\n'){ /* newline */
            ln++;
            col = 0;
        }
        else if (*s == '\f'){ /* form feed (clr screen) */
            ln = 0;
            col = 0;
            display_clear(Black);
        }
        else if (*s == '\r'){ /* return */
            col = 0;
        }
        else{
            if(col >= (width/font_width)){
                ln++;
                col = 0;
            }
            if(ln >= height/font_height){
                ln = height/font_height-1;
                ScrollVertical(font_height);
            }
            display_print_char (ln, col++, *s);
        }
        s++;
    }
}

/*---------------------------------------------------------------------------
  Set the text color
    - color: Color
 *---------------------------------------------------------------------------*/
void display_set_text_color (unsigned short color) {
  Color[TXT_COLOR] = color;
}

/*---------------------------------------------------------------------------
  Set the background color
    - color: Color
 *---------------------------------------------------------------------------*/
void display_set_back_color (unsigned short color) {
  Color[BG_COLOR] = color;
}

/*---------------------------------------------------------------------------
  Set if display is landscape or portrait
    - ls: 0: Portrait, 1: Landscape
 *---------------------------------------------------------------------------*/
void display_set_landscape (uint8_t ls){
    landscape = ls;
    if(ls){
        width = PHYS_YSZ;
        height = PHYS_XSZ;
    }
    else{
        width = PHYS_YSZ;
        height = PHYS_XSZ;
    }
}

/*---------------------------------------------------------------------------
  Set the font size
    - font: SMALL_FONT or BIG_FONT
 *---------------------------------------------------------------------------*/
void display_set_font_size (unsigned short font) {
    switch(font){
        case SMALL_FONT:
            font_size = 0;
            font_width = SMALL_FONT_WIDTH;
            font_height = SMALL_FONT_HEIGHT;
            break;
        case BIG_FONT:
            font_size = 1;
            font_width = BIG_FONT_WIDTH;
            font_height = BIG_FONT_HEIGHT;
            break;
        default:
            font_size = 0;
            font_width = SMALL_FONT_WIDTH;
            font_height = SMALL_FONT_HEIGHT;
            break;
  }
}

/*---------------------------------------------------------------------------
  Initialize the on-board display
 *---------------------------------------------------------------------------*/
void display_init(void){
    // Board spesific init
    Board_LCD_Init();
    // Clear frame buffer
    display_clear(Black);
    //delay(5);
    // Turn on LCD
    Chip_LCD_Init( (LCD_Config_Type *) &BOARD_LCD);
    Chip_LCD_SetUPFrameBuffer( (void *) framebuffer);
    Chip_LCD_Power(ENABLE);
    delay(5);
    Board_SetLCDBacklight(1);
    // Set text color
    display_set_font_size(SMALL_FONT);
    display_set_back_color(Black);
    display_set_text_color(Green);
}

/*---------------------------------------------------------------------------
  Implementation of sendchar function used in Retarget.c
 *---------------------------------------------------------------------------*/
int sendchar(int ch){
    static unsigned int ln = 0;
    static unsigned int col = 0;

    char c = (char) ch;
    if(c == '\n'){ /* newline */
        ln++;
        col = 0;
    }
    else if (c == '\f'){ /* form feed (clr screen) */
        ln = 0;
        col = 0;
        display_clear(Black);
    }
    else if (c == '\r'){ /* return */
```

```
            col = 0;
        }
        else{
            if(col >= (width/font_width)){
                ln++;
                col = 0;
            }
            if(ln >= height/font_height){
                ln = height/font_height-1;
                ScrollVertical(font_height);
            }
            display_print_char(ln, col, (char) c);
            col++;
        }
        return (int) ch;
}

/*----------------------------------------------------------------------------
  Implementation of getkey function used in Retarget.c
 *----------------------------------------------------------------------------*/
int getkey(void){return 0;}
```

Listing 6: display_mcb4300.c

A.7 Spectrum Analyzer

```c
#include "spectrum.h"
#include "board.h"
#include "memory_map.h"
#include "display_mcb4300.h"
#include "uda1380_mcb4300.h"
#include "dsp_funcs.h"
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

// Remove sprintf not compatible with char* warning
#pragma diag_suppress 167

// Spectrum Parameters
#define SPECTRUM_SCREEN_X_SIZE 320
#define SPECTRUM_SCREEN_Y_ZIZE 240
#define SPECTRUM_HEIGHT 210
#define SPECTRUM_WIDTH 312
#define SPECTRUM_START_Y (SPECTRUM_HEIGHT + 15)
#define SPECTRUM_START_X (SPECTRUM_SCREEN_X_SIZE - SPECTRUM_WIDTH)

// Colors used by spectrum
#define SPECTRUM_BACK_CLR Black
#define SPECTRUM_TEXT_CLR Cyan
#define SPECTRUM_POINT_CLR Orange

// Parameters for Welch estimate
#define WELCH_N 20480
#define WELCH_OVERLAP 1536

// Private variables
static float valsprbin;
static uint32_t num_vals;

// Strings
static const char title_s[] = "AUDIO SPECTRUM ANALYZER";

// Shared variables
volatile float maxval = 1.0f;
int adc_on = 0;
int spectrum_on = 0;

/*-----------------------------------------------------------------------------
  Draw the point on the spectrum by coloring a column of pixels ut to the
  correct point, forming a bar. (Float version)
  Will convert to dB scale with max_val as the reference value.
    - val: The value of the point (float)
    - xpos: The x position of the point
 *----------------------------------------------------------------------------*/
static void putPointFl(float val, uint16_t xpos){
    static int16_t ypos;

    if(xpos < SPECTRUM_WIDTH){
        // Convert to dB.
        if(val == 0)
            val = -9999;
        else if(val < maxval)
            val = 10*log10(val/maxval);
        else val = 0;
        // Convert to y position on screen
        ypos = (val + 80) * SPECTRUM_HEIGHT/80;
        if (ypos <= 0) ypos = 0;
        // Draw the spectrum lines
        display_draw_line(SPECTRUM_SCREEN_X_SIZE-SPECTRUM_START_X-xpos,
            SPECTRUM_START_Y,ypos,0,SPECTRUM_POINT_CLR);
        display_draw_line(SPECTRUM_SCREEN_X_SIZE-SPECTRUM_START_X-xpos,
            SPECTRUM_START_Y-ypos,SPECTRUM_HEIGHT-ypos,0,SPECTRUM_BACK_CLR);
    }
}

/*-----------------------------------------------------------------------------
```

```
  Display the recorded spectrum on the screen. When there is more values than
  pixels, it will display the averages of adjacent samples. (Float version)
    - vals: Pointer to the array containing the magnitude values
    - num_vals: Number of values in the array
 *--------------------------------------------------------------------------*/
void spectrum_display_spectrum_f32(float32_t *vals){
    static float i;
    static float j;
    uint16_t k = 1;
    float bin = 0;
    // Join adjacent values into bins to fit the pixels on the display
    if (valsprbin >= 1) {
        j = 0;
        for (i = 0; i < num_vals; i++){
            bin += vals[(uint32_t)i];
            j++;
            if((i+1.0f) / valsprbin >= k){
                bin /= j;
                putPointFl(bin,k-1);
                bin = 0;
                j = 0;
                k++;
            }
        }
    }
    else{
        //TODO: Bars larger than 1 pixel..
    }
}

/*---------------------------------------------------------------------------
  Initialize the spectrum analyzer
    - fs: Sampling frequency
 *--------------------------------------------------------------------------*/
void spectrum_init(uint16_t fs, uint32_t bins){
    static int i;
    char buf[64];
    static ADC_Clock_Setup_Type ADCSetup;

    // Calculate number of values per bin
    valsprbin = (float)bins / (float)SPECTRUM_WIDTH;
    num_vals = bins;

    // Set up the display
    display_clear(SPECTRUM_BACK_CLR);
    display_set_back_color(SPECTRUM_BACK_CLR);
    display_set_text_color(SPECTRUM_TEXT_CLR);
    display_set_font_size(SMALL_FONT);
    display_set_landscape(1);
    display_print_string(0,0,title_s);
    display_print_string((240/8)-1,1,"0 kHz");
    sprintf(buf, "%3.1f", (float)fs/4000);
    display_print_string((240/8)-1,(320/12),buf);
    sprintf(buf, "%2.0f", (float)fs/2000);
    display_print_string((240/8)-1,(320/6)-2,buf);
    display_draw_line(0,10,320,1,SPECTRUM_TEXT_CLR);
    display_draw_line(0,229,320,1,SPECTRUM_TEXT_CLR);
    for(i = 0; i <= SPECTRUM_WIDTH; i += 13){
        display_draw_line(i,SPECTRUM_START_Y+5,3,0,SPECTRUM_TEXT_CLR);
    }

    // ENABLE ADC TO CONTROL MAX LEVEL SENSITIVITY
    Board_ADC_Init();
    Chip_ADC_Init(LPC_ADC0, &ADCSetup);
    Chip_ADC_Channel_Enable_Cmd(LPC_ADC0, ADC_CH1, ENABLE);
    Chip_ADC_Set_Resolution(LPC_ADC0, &ADCSetup, ADC_10BITS);
    Chip_ADC_Set_SampleRate(LPC_ADC0, &ADCSetup, 400000);
    // Enable ADC Interrupt
    NVIC_ClearPendingIRQ(ADC0_IRQn);
    NVIC_EnableIRQ(ADC0_IRQn);
    Chip_ADC_Channel_Int_Cmd(LPC_ADC0, ADC_CH1, ENABLE);

    // Start the codec and set flags for SysTick handler
    uda1380_start_stream();
    adc_on = 1;
    spectrum_on = 1;
}// end spectrum_init()

/*---------------------------------------------------------------------------
  De-Initialize the spectrum analyzer, i.e. stop the AD converter
 *---------------------------------------------------------------------------*/
void spectrum_uninit(void){
    // Stop the codec
    uda1380_stop_stream();

    // Stop the ADC
    NVIC_DisableIRQ(ADC0_IRQn);
    Chip_ADC_Channel_Int_Cmd(LPC_ADC0, ADC_CH1, DISABLE);
    Chip_ADC_Channel_Enable_Cmd(LPC_ADC0, ADC_CH1, DISABLE);
    Chip_ADC_DeInit(LPC_ADC0);

    // Clear flags for SysTick handler
    adc_on = 0;
    spectrum_on = 0;
}

/*---------------------------------------------------------------------------
  Make an audio spectrum by taking a Welch estimate and displaying
  the resulting spectrum on screen
 *---------------------------------------------------------------------------*/
void spectrum(void){
    static float32_t *res;

    // Get Welch estimate
    res = dsp_welch(WELCH_N, WELCH_OVERLAP);

    // Display the magnitudes as a spectrum
    spectrum_display_spectrum_f32(res);
}

/*---------------------------------------------------------------------------
  IRQ handler for the ADC0 to control the reference (max) value of the spectrum
 *---------------------------------------------------------------------------*/
void ADC0_IRQHandler(void){
    uint16_t dataADC;
    float32_t new_val;

    NVIC_DisableIRQ(ADC0_IRQn);
    Chip_ADC_Channel_Int_Cmd(LPC_ADC0, ADC_CH1, DISABLE);

    // Read ADC value
    Chip_ADC_Read_Value(LPC_ADC0, ADC_CH1, &dataADC);

    // Calculate new max value
    if(dataADC != 0){
        new_val = (float)dataADC/1024.0f;
        if((new_val > (maxval + 1.0f/40.0f))
            || (new_val < (maxval - 1.0f/100.0f))){
            maxval = new_val;
        }
    }
    NVIC_EnableIRQ(ADC0_IRQn);
    Chip_ADC_Channel_Int_Cmd(LPC_ADC0, ADC_CH1, ENABLE);
}
```

Listing 7: spectrum.c

B Matlab Scripts

B.1 Simulation of Direct Welch Method

```
O = 0; % Overlap in percent
M = 2048; % Length of M
D = M * O/100; % Number of samples overlap
N = 20480; % Total number of samples
ymin = 0; % Min y-value for plot
ymax = 50; % Max y-value for plot

fname='0-50-0-low.wav'; % File to read

%Read the file
[x,fs]=wavread(fname);

% Filter the signal with FIR high-pass filter
fc = 160; % Cutoff frequency
B= fir1(100,fc*2/11025,'high');
A = 1;
xfilt = filter(B,A,x);

% Set up arrays and variables
v = [];
j = 1;
f = 0 : 11025/1024 : 11025;
ff = 0 : 0.5 : 249.5;

% Go through the file and do measurements at N/2 intervals
for i = 1:N/1:length(x)-N
    % Fine Welch estimate
    Pxx = pwelch(xfilt(i:i+N),hamming(M),D,M,fs);

    % Find highest compontent of estimate
    [m, index] = max(Pxx);
    fspeed = (index / M) * fs;
    % Calculate speed based on highest component
    v(j) = (fspeed*3e8)/(2*24.1e9*cos(pi/4)) * 3.6;

    % Uptate index for next measurement
    j = j+1;
end

% Plot measured speeds
figure(1);
t = 1:j-1;
plot(t,v,'*r');
grid('on');
ylim([ymin,ymax]);
xlim([0 length(v)]);
xlabel('Measurement number');
ylabel('Estimated Speed (km/h)');
title('Strongest Component from Welch Estimate');
```

Listing 8: simulation_welch.m

B.2 Simulation of Correlation Method

```matlab
O = 75; % Overlap in percent
M = 2048; % Length of M
D = M * O/100; % Number of samples overlap
N = 20480; % Total number of samples
ymin = 0; % Min y-value for plot
ymax = 50; % Max y-value for plot

fname='0-50-0-low.wav'; % File to read

%Read the file
[x,fs]=wavread(fname);

% Filter the signal with FIR high-pass filter
fc = 160; % Cutoff frequency
B= fir1(100,fc*2/11025,'high');
A = 1;
xfilt = filter(B,A,x);

% Set up arrays and variables
v2 = [];
Cs = zeros(1,400);
j = 1;
f = 0 : 11025/1024 : 11025;
ff = 0 : 0.5 : 249.5;

% Go through the file and do measurements at N/2 intervals
for i = 1:N:length(x)-N
    % Fine Welch estimate
    Pxx = pwelch(xfilt(i:i+N),hamming(M),D,M,fs);

    % Compare Pxx and Pyy
    for v1 = 1 : 1 : 500
        Pyy = rot90(Pfv(v1,1:1025),1);
        C = conj(Pxx) .* Pyy;
        Cs(v1) = sum(C);
    end
    % Calculate speed baset on strongest correlation
    [m, index] = max(Cs);
    v2(j) = (index-1)/2;

    % Uptate index for next measurement
    j = j+1;
end

% Plot measured speeds
figure(1);
t = 1:j-1;
plot(t,v2,'*r');
grid('on');
ylim([ymin,ymax]);
xlim([0 length(v2)]);
xlabel('Measurement number');
ylabel('Estimated Speed (km/h)');
title('Strongest Spectral Correlation');
```

Listing 9: simulation_correlation.m

B.3 Estimation of Theoretical Doppler Spectra

```
Pt = 5e-3;  % Transmit power
L = 0.01245; % Wavelength
c0 = L^2 * Pt * (4*pi)^-3; % Radar equation constant

num_speeds = 500; % Number of speeds to estimate
fmax = 11025; % Max doppler frequency to estimate

h = 0.5; % Radar HoG
tilt = 45; % Sensor tilt in degrees
hwidth = 72; % horizontal 3dB beam with (72 deg. from datasheet)
vwidth = 18; % vertical 3dB beam with (18 deg. from datasheet)

% Illumination Area depth (3db bandwidth)
Ad = h/tan((tilt-vwidth/2)*pi/180) - h/tan((tilt+vwidth/2)*pi/180);
% Illumination Area width (3db bandwidth)
Aw = h/sin(tilt*pi/180) * tan((hwidth/2)*pi/180) * 2;
A = Ad * Aw; % Approx. illuminated area

f = 0 : fmax/1024 : fmax; % Number of frequencies to plot

Pfv = zeros(num_speeds,size(f,2));

for v = 0.5 : 0.5 : 249.5
    v0 = v/3.6;
    for i = 1 : size(f,2)
        % Find number of values to estimate
        am = real(acos(f(i)*(L/(2*v0))));
        if am == 0
            num = i;
            break
        end
    end
    a = real(acos(f.*(L/(2*v0)))); % Angles
    a = a.*180/pi; % To degrees

    % Estimate power specrtrum
    for i = 1 : num
        % Find radar cross section
        sigma0 = findS0(a(i));
        sigma = sigma0 * A;

        % Find the antenna gain from antenna diagram (corrected for tilt)
        phi = findPhi(a(i)+90-tilt);

        % Find the distance to the road
        r = h/sin(a(i)*pi/180);

        % Calculate the power distribution
        Pfv(v*2+1,i) = (1/num)*(c0 * sigma * phi^2) / r^4;
    end
    Pfv(v*2+1,1:1025) = rot90(Pfv(v*2+1,1:1025),2);
end

% Write the estemated spectra to file used by microcontroller
fid = fopen('pyy.bin', 'w');
fwrite(fid, rot90(Pfv(1:num_speeds,1:1025)), 'float32',0,'l');
fclose(fid);
```

Listing 10: pyy_estimation.m

B.4 Plot Measurements vs GPS reference including errors

```
% Define scale and limits:
scale = 1.20;
limit = 5;
uplimit = 70;

% Read measurement file
fid = fopen('5.DOP', 'r');
% Timestamp is first 16 bytes
timestamp = fread(fid,4,'uint32',0);
% Rest is measurement values
values = fread(fid,inf,'float32',0,'l');
fclose(fid);

% Read GPS file
fid = fopen('result.txt', 'r');
SPD = textscan(fid, '%s %s %f32');
fclose(fid);

% Find start and end time for GPS results
formatIn = 'HHMMSS ddmmyy';
datestring = sprintf('%s %s',SPD{1}{1},SPD{2}{1});
starttime = datenum(datestring,formatIn);
datestring = sprintf('%s %s',SPD{1}{end},SPD{2}{end});
endtime = datenum(datestring,formatIn);
seconds = etime(datevec(endtime),datevec(starttime));

% Plot GPS values
fig = figure(1);
subplot(2,1,1);
x = 0:seconds;
p1 = plot(x,SPD{3},'bo');
hold('on');

% Bitmask
mask = hex2dec('000000FF');

% Measurement type
type = bitshift(timestamp(1,1),-24);
switch type
    case 0
        type_string = 'PERIODOGRAM';
    case 1
        type_string = 'WELCH ESTIMATE';
    case 2
        type_string = 'CORRELATION';
    otherwise
        type_string = '';
end

% Measurement start time
shour = bitand(bitshift(timestamp(1,1),-16),mask);
smin = bitand(bitshift(timestamp(1,1),-8),mask);
ssec = bitand(bitshift(timestamp(1,1),-0),mask);
syear = bitshift(timestamp(2,1),-16);
smonth = bitand(bitshift(timestamp(2,1),-8),mask);
sday = bitand(bitshift(timestamp(2,1),-0),mask);
start = datenum(0,0,0,shour,smin,ssec);

% Measurement end time
fhour = bitand(bitshift(timestamp(3,1),-16),mask);
fmin = bitand(bitshift(timestamp(3,1),-8),mask);
fsec = bitand(bitshift(timestamp(3,1),-0),mask);
fyear = bitshift(timestamp(4,1),-16);
fmonth = bitand(bitshift(timestamp(4,1),-8),mask);
fday = bitand(bitshift(timestamp(4,1),-0),mask);
finish = datenum(0,0,0,fhour,fmin,fsec);

% Real Measurement start and end time
meas_start = starttime + start;
```

```
meas_end = starttime + finish;

% Number of measurements
num = length(values)-1;
zerotime = datenum(0,0,0,0,0,0);

% Elapsed time in seconds
seconds = etime(datevec(meas_end),datevec(meas_start));
startx = etime(datevec(meas_start),datevec(starttime));
endx = startx + seconds;

% Ignore values of 108 (should be 0).
for i = 1 : length(values)
    if values(i) == 108
        values(i) = 0;
    else
        values(i) = values(i)*scale;
    end
end

% Find Downsampling fraction
Fs = num/seconds;
R = round(Fs * 1000);

% Downsample to 1 Hz to align with GPS results
upvalues = interp(values,1000);
yvalues = decimate(upvalues,R);

lag = 1; % Measurement started 1 second after GPS
x = startx+lag:1:endx+lag;
xlimit = [startx endx];

% Plot Aligned Values
p2 = plot(x,yvalues,'r*');
xlabel('Seconds since 03-Jun-2013 12:33:14 (UTC)');
ylabel('Speed (km/h)');
grid('on');
xlim(xlimit);
title(['Aligned Speed Measurements. Start time: ' datestr(meas_start) ', End time: '
      datestr(meas_end)]);
hold('off');
leg1 = legend([p1,p2],'GPS','Radar');
set(leg1,'Location','NorthEast')

% Calculate and plot error in km/h and %
error1 = [];
error2 = [];
erri = 1;
for i = xlimit(1) : xlimit(2)
    if( SPD{3}(i+lag) < limit || yvalues(i-startx) < limit*scale || SPD{3}(i+lag) >
         uplimit)
        %error(i-startx) = 0;
    else
        error1(erri) = (yvalues(i-startx) - SPD{3}(i+lag));
        error2(erri) = 100*(yvalues(i-startx) - SPD{3}(i+lag))/SPD{3}(i+lag);
        erri = erri+1;
    end
end
m1 = sprintf('%f',mean(error1));
s1 = sprintf('%f',std(error1));
m2 = sprintf('%f',mean(error2));
s2 = sprintf('%f',std(error2));

subplot(2,2,3);
hist(error1,50);
grid('on');
ylabel('Number of measurements');
xlabel('Error in km/h');
title(['Mean: ' m1 ', Std: ' s1]);

subplot(2,2,4);
hist(error2,50);
grid('on');
ylabel('Number of measurements');
```

```
xlabel('Error in %');
title(['Mean: ' m2 ', Std: ' s2]);

% Save plot to PNG file
name = sprintf('%d_%d_total_corr2',xlimit(1),xlimit(2));
set(fig,'PaperUnits','inches','PaperSize',[10,6],'PaperPosition',[0 0 10 6]);
print('-dpng','-r300',name);
```

Listing 11: plot_all.m