NTNU
Norwegian University of Science and Technology

A Modular Software and Hardware Framework with Application to Unmanned Autonomous Systems

Interacting Modules, Error Detection and Hardware Design

Kristoffer Rist Skøien

Master of Science in Engineering Cybernetics

Submission date:	June 2011
Supervisor:	Amund Skavhaug, ITK
Co-supervisor:	Thor Inge Fossen, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem Description

In 2010 the Unmanned Vehicle Laboratory was established at the Department of Engineering Cybernetics. Assignments will continue where the fall projects left off, to take one step closer to fully autonomous flight.

The candidate shall make use of the existing work to achieve a system that can be a base for further development and testing.

It is of importance that functionality is hidden to a high extent, and solutions chosen so that the system can be easily modified, ensuring that new candidates can continue the work with ease and making the system long-lived.

Three objectives are given:
- Create a modular software system that can complement the modularity of the hardware. Advanced functionality should be hidden, and as high level of modularity and confinement achieved as possible.
- A basic and general sensor alert system must be made. Basic faults in sensors such as no signal or stationary signal must be detected to make sure that control algorithms can take appropriate action.
- Hardware must be interfaced to actuators and other sensors applicable in the UAV domain.

Experiences shall be drawn from related work, found solutions must be implemented, central decisions commented and evaluated, and hardware designed if necessary.

Assignment given: January 17, 2011

Student: Kristoffer Rist Skøien
Supervisor: Associate Professor Amund Skavhaug, Dept. of Engineering Cybernetics, NTNU
Co-supervisor: Professor Thor I. Fossen, Dept. of Engineering Cybernetics, NTNU

A Modular Software and Hardware Framework with Application to Unmanned Autonomous Systems

Interacting Modules, Error Detection and Hardware Design

Kristoffer Rist Skøien

Master's Thesis

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

June 19, 2011

Extended Abstract

The Department of Engineering Cybernetics at the Norwegian University of Science and Technology established the Unmanned Vehicle Laboratory fall 2010. The goal is to have students develop a fully functional autonomous aerial vehicle over time as part of projects and master's theses. A range of projects were carried out fall 2010, among others the report *General Platform for Unmanned Autonomous Systems* was written on the topic of hardware, operating systems and peripheral interfacing.

This master's thesis continues where the previous report left suggestions for further work, and covers the topics of a software framework, sensor error detection, actuator and sensor interfacing, that is now part of the autonomous flight system.

A highly modular software framework has been constructed, applicable far beyond the unmanned vehicle domain. Due to the high level of encapsulation and modularity it is especially valuable in projects where there is a high mobility of the workforce, such as student projects and theses. It acts as a middleware layer, with language independent, separately compilable modules, communicating with one another to achieve the desired functionality.

To prevent unrealistic or erroneous sensor readings from spreading through the system, a software detection unit catches signal anomalies based on statistics and alerts subscribing modules. The algorithm has been interfaced into the software framework, and is applicable to numerous sensors.

Hardware was designed, constructed and tested to handle sensor interfacing, power supply demands and real-time critical actions such as actuator control. The design is performed from an aerial vehicle application point of view, but general to such an extent that is usable in a wide range of autonomous crafts.

The framework, filter and hardware are merged together and tested on an embedded system, verifying the system functionality with a feedback loop from measurements to actuators. Utilizing the previous work along with all three elements of this thesis, a fully functional system for vehicle control is achieved.

Preface and Acknowledgements

The following master's thesis is the final contribution to achieve the title *Master of Science* from the Norwegian University of Science and Technology at the Department of Engineering Cybernetics.

It has been inspiring to be among the first batch of students utilizing the Unmanned Vehicle Laboratory, and take the first steps in making the schools own unmanned aerial vehicle. I believe that the effort put into all the projects and theses through the last year will provide a solid foundation for further progress for years to come.

It has at times been difficult, dealing with unknown issues which most of the time results in tasks taking much more time than planned. One must not be deterred by this, and I will let this thesis be a testimony to what can be done and challenges overcome by drawing on help from others and putting in the necessary effort. Most of the time it has been a pleasure to write this thesis due to its nature in the intersection between the software and hardware realm, where I consider myself most at home.

It is my great hope that the unmanned aerial vehicle can one day soar with agility and grace through the skies, being an inspiration for both current and potentially new students at the faculty of Engineering Cybernetics.

I give my heartfelt thanks to my supervisor, Associate Professor Amund Skavhaug for his profound help and my co-supervisor and head of the Unmanned Vehicle Laboratory, Professor Thor Inge Fossen for his assistance.

A big thank you to the friends i have been fortunate enough to have and gotten to know through these years. They are the ones who made this period of my life to something truly special.

I would also like to thank my entire family and especially my parents; Hilde and Espen and my brother Lasse for always being there for me.

Finally, I would like to thank my girlfriend Kristina, for being so kind, supportive and encouraging through this final semester.

Trondheim, June 19, 2011

Kristoffer Rist Skøien

Contents

List of Figures

List of Tables

Notation, Nomenclature and Abbreviations

Notation

x	-	Normal font indicates a scalar variable
$\mathbf{X}$	-	Bold font indicates a vector or a matrix
`output`	-	Style indicates short examples of system output or indicates a variable or a function
1 `output`	-	Indicates source code, line numbers staring with 1 Similar formatting without line numbering indicates longer examples of system output
$\hookleftarrow$	-	Line wrap due to paper width
Italic font	-	Italics are used on names or to emphasize

Nomenclature

x	-	Position or axis x
y	-	Position or axis y
z	-	Position or axis z
ϕ	-	Roll in Euler angle
θ	-	Pitch in Euler angle
ψ	-	Yaw in Euler angle
θ_S	-	Rotational position of servo
e	-	Error
r	-	Reference
u	-	Input
y	-	Output
F_{CPU}	-	CPU frequency
C_{TOP}	-	Counter top value
C_{OVF}	-	Counter overflow value
t_p	-	Pulse period

Abbreviations

ADC	-	Analog Digital Converter
BB	-	BeagleBoard
CG	-	Center of Gravity
DAC	-	Digital Analog Converter
DMA	-	Direct Memory Access
DoF	-	Degrees of Freedom
EEPROM	-	Electrically Erasable Programmable Read-Only Memory
ESC	-	Engine Speed Control
GNC	-	Guidance Navigation and Control
GPIO	-	General Purpose Input Output
GNU	-	Gnu's Not Unix
I/O	-	Input/Output
IDE	-	Integrated Development Environment
IMU	-	Inertial Measurement Unit
IPC	-	Inter-Process Communication
ITK	-	Department of Engineering Cybernetics (Institutt for teknisk kybernetikk)
LiPo	-	Lithium Polymer
LDO	-	Low-dropout
MCU	-	Microcontroller Unit
OS	-	Operating System
PAL	-	Programmable Array Logic
PCB	-	Printed Circuit Board
PDI	-	Program and Debug Interface
PID	-	Process ID
POSIX	-	Portable Operating System Interface for Unix
PWM	-	Pulse Width Modulation
RC	-	Radio Controlled
UAV	-	Unmanned Aerial Vehicle
UVL	-	Unmanned Vehicle Laboratory
UV	-	Unmanned Vehicle

Chapter 1

Introduction

1.1 Motivation

Unmanned vehicles started to appear along with advances in technology that allowed for remote operation. The application for unmanned vehicles stretches across many domains such as on water, under water, on land and in the air, in vast and microscopic scales. Uses span an immense specter where it is dangerous, inefficient or impossible for humans to venture. As advances in science are made, technologies that before were restricted due to price or secrecy are becoming more available. This is particularly the case for unmanned vehicles where autonomous control used to be a costly procedure, due to expensive sensor, radio and imagery technology. At present time, reasonably priced technology can be bought from a range of suppliers providing an easy and low-cost entrance to the world of unmanned vehicles.

In the field of cybernetics, this is a phenomenal area to explore as cybernetics can be utilized to a great extent to control these vehicles. Schools and private companies have indeed recognized this field as a great area to learn, give assignments and develop technology which can turn a profit.

1.2 The Big Scope

The Department of Engineering Cybernetics established the "Unmanned Vehicle Laboratory" (UVL) fall 2010. This lab serves as a hub for students performing assignments that falls under this category, and encourages teamwork, sharing of information and a better environment for UV development purposes. Assignments have for several years been carried out at the department in the field of unmanned vehicles, but the projects tend to halt when students graduate. One of the initial physical objectives is to develop an in-house, low cost, fully automated unmanned aerial system in a short amount of time. The UAV will serve many purposes such as PhD, master and project assignments over many years, be used as a test facility for new components, gathering of test data, be part of international unmanned aerial vehicle (UAV) competitions and be used as advertisement.

1.3 Previous Work

Since the UV lab started in 2010, a range of projects and master's theses have been written on the subject. With the in-house UAV in mind, consider Figure 1.1 for what has been executed so far. Blue indicates contents of this thesis. See Section 1.4 for details. Green is already addressed issues that are completed to such a degree that they can support the vehicle in autonomous flight. As is ordinary with prototypes and projects in development, all parts are subject to continuous improvement and alteration. Thus should no part be considered as completely finished. Manual control is marked as addressed since this is supplied with the currently used aircraft (The Recce D6 from Odin Aero AS). The inertial measurement unit (IMU) is also marked green due to the fact that this only needs minor alterations to become part of the framework. This is also the case for the GPS which is fully operable, but interfaced to the previous framework. White boxes are unaddressed issues.

Assignments performed so far in the UVL targeted towards the Recce D6 are summarized here.

- The assignment regarding motherboard, OS, servo-control and GPS was addressed in [1, Skøien, Vermeer].
- A model of the Recce D6 and a simulator can be found in [2, Dønnestad pp.34-46 and pp.22-33 respectively], and in [3, Dønnestad].
- Several observers have been made, and are discussed in [4, Ingebretsen] [5, Nøkland] [6, Hafslund] and [7, Nøkland].
- Work regarding hardware in the loop has been under development spring 2011 and can be found in [8, Stern].

1.4 Main Contributions and Contents of Thesis

This thesis contributes in two directions at once. One very general direction, producing solutions that are usable in many domains and one targeted towards completing the specific UAV described. Many of the results in this thesis are general to such an extent that they are applicable far beyond the unmanned laboratory and unmanned vehicles, making the thesis of interest to a wide audience within the genre of embedded solutions and computer science. Specific implementations, figures, some tests and examples are targeted towards UAVs and the Recce D6 platform to set the solutions into context. By utilizing and building upon the work that has already been done, and then performing further studies, construction and implementation, this thesis takes the Recce D6 a large step closer to autonomous flight.

The advantage of viewing this thesis along side with the underlying project report [1, Skøien, Vermeer] *General Platform for Unmanned Autonomous Systems* must be emphasized, which founded much of the base for further work. It is included in the digital appendix A.

The target audience are individuals with knowledge in the field of cybernetics, programming, electronics, unmanned vehicles and some mathematics, especially future UAV-lab participants. The thesis may also interest general embedded and code developers, students and radio control hobbyists.

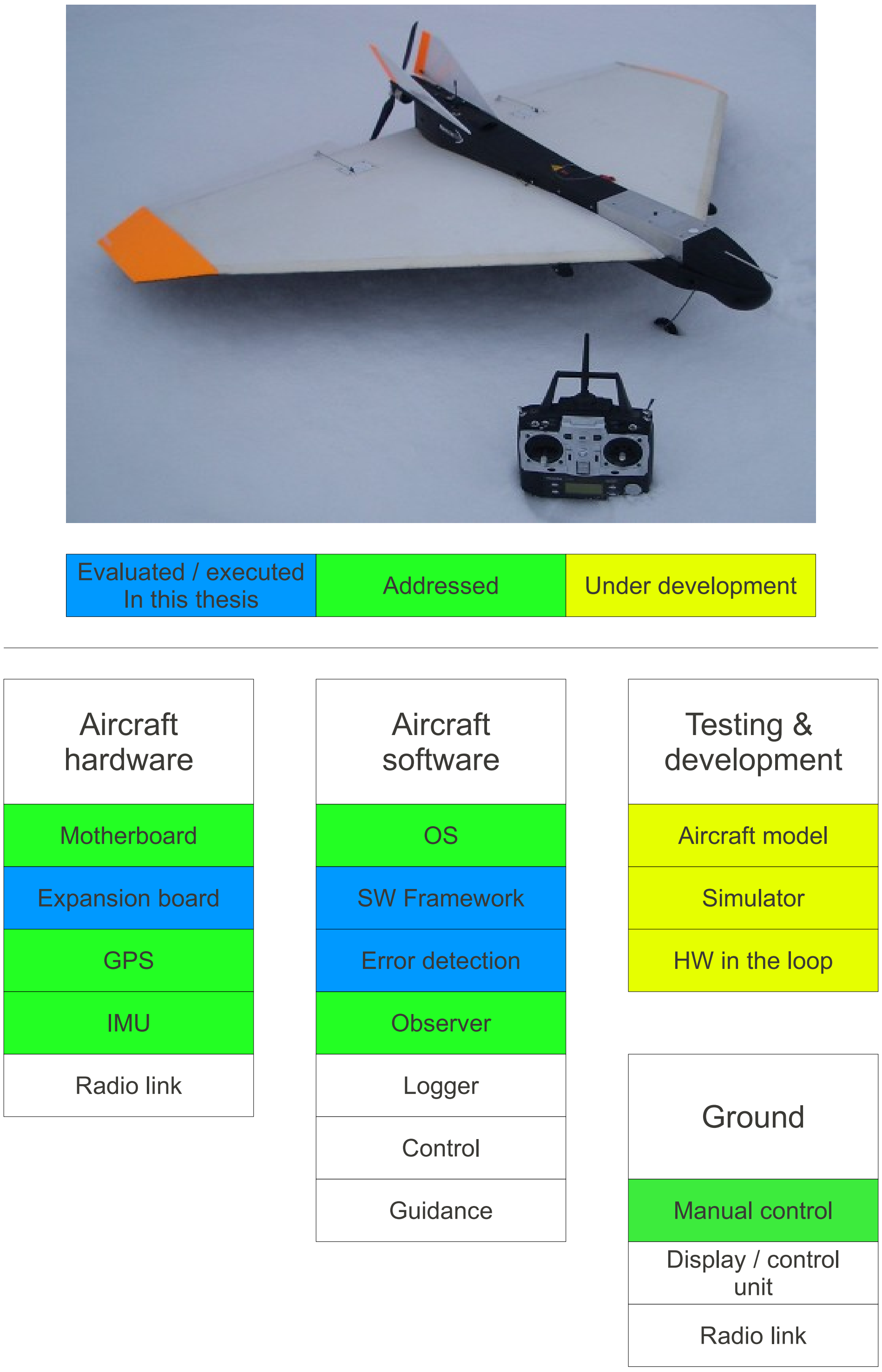


Figure 1.1: Project overview with the Recce D6. Picture courtesy of Odin Aero AS

To complete the platform, the following issues are considered in this thesis:

- SW framework
  A highly modular framework is constructed, supporting modules in a variety of programming languages. The software is able to accommodate the modularity of the hardware, detect faulty modules, and take action to correct. Based on extracted work form this thesis, a paper [9, Skøien, Skavhaug] on the software topic is being

prepared for publication by the author and main supervisor.

- Sensor error detection
  The need for an adaptable low-level sensor error detection module is addressed. This will alert modules such as observers and control systems of faulty sensor readings, and prohibit these from going astray. By looking at such parameters as variance, some information regarding the quality of the signal is also known.

- Support hardware and servo control
  To control actuators, provide extra safety, monitor power usage and provide a stable power supply, an add-on board is designed and constructed. A range of sensors interface to this board, becoming a part of the total framework.

This thesis provides a general software and hardware solution, applicable to a wide range of unmanned crafts. Due to the high level of modularity and confinement, the solution is especially applicable in projects and workplaces where there is a high mobility of the workforce, such as school projects and theses. By reading further, one will be presented with solutions applicable to a range of unmanned vehicles, as well as the software framework being applicable in general software and embedded solutions.

As one can observe from Figure 1.1, by providing these additional pieces, the puzzle will become complete to such an extent that autonomous flight is possible, except for guidance and control. These two issues have yet to be addressed, so for initial test purposes the guidance system will be omitted, and a simple P regulator used for vehicle control. The radio link and logger is not necessary to fly autonomously.

1.5 Thesis Structure

The contents of this thesis as noted in Section 1.4 is presented by focusing on to two main topics which are critical to complete the system for autonomous flight, and the signal detection algorithm which is a smaller topic with less focus. Finally, all three topics are tested together.

Chapter 2, Software Framework
This chapter describes the work surrounding the software framework, code examples and extended descriptions of the system are given.

Chapter 3, Sensor Signal Quality and Error Detection
An IMU is interfaced to Simulink for system debugging and visual reference, and a general basic signal error detection system is coded. The desire and need for a universal solution results in this chapter being somewhat briefer than the other two main chapters due to the generic implementation.

Chapter 4, BeagleBoard UAV Support Hardware
Demands, design, development and test of an expansion board is covered here.

Chapter 5, Full System Test
This chapter executes a full system test, demonstrating feedback functionality with all three previous chapters functioning together.

Chapter 6, Overall Discussion and Evaluation
Discussions are handled in their respective chapter. A brief overall discussion is found here as well as a reflection on the performed work.

Chapter 7, Future Work
Recommendations regarding alterations and further work are summarized in this chapter.

Chapter 8, Conclusion
The final conclusion resides here.

The nature of this thesis suggests that studies, demands, results and discussions, are part of each separate chapter. A short general discussion and conclusion is located at the end to maintain readability. For that same reason, some guides, tables and procedures have been moved to the appendix. This way, developers interested in performing a specific action can use a particular appendix in isolation as a guide. All figures are created by the author unless otherwise stated in the caption. References are made to specific pages if applicable, to simplify the process of finding source material.

Chapter 2

Software Framework

2.1 Background

In projects within the domain of computer science, modularity is an important issue. Consider a project with a certain number of people are involved in code production. Modular code lets developers define interfaces between sections of code (e.g. function calls), and then focus on developing and debugging their specific section. This also allows for code reuse, as one section of code performs a certain operation that can be eligible in many places. In the area of unmanned vehicles (UV), it has for long been a common practice to use a microcontroller unit (MCU) to handle vehicle control in all regards, due to the low weight, cost, power consumption and price. Since microcontrollers often impose quite some computational limitations, such as: limited core frequency, little memory, "big while" implementation, only one priority level and no multithreading capabilities, the code must be designed accordingly. This can in some projects result in highly integrated code. As pointed out in [10, Garcia et al. pp.96] this have several disadvantages:

- Removing obsolete code is difficult. The highly integrated code may have connections to many parts of the total structure.
- Expansions and upgrades are hard to perform as one needs more extensive knowledge of the system.
- In highly integrated code, timing and synchronization with external devices becomes rigid. On the other hand in a multi threaded environment, a thread can be scheduled out while waiting for an external unit and thus free valuable time for other applications.
- Testing after upgrades and/or alterations is more comprehensive. Consider a single "threaded" MCU application vs. a multithreaded PC application. An addition to the first causes the entire program to increase its loop-time whilst this might not be the case in the latter.

As development progresses, more powerful systems are developed in smaller packages at a more reasonable price. In this particular project the BeagleBoard (BB)(See Figure 2.1) is chosen as the vehicle's main control unit, running Ångström Linux, providing extraordinary computing capabilities in a small package. This enables the developers to avoid some of the issues that arise from ordinary microcontroller applications.

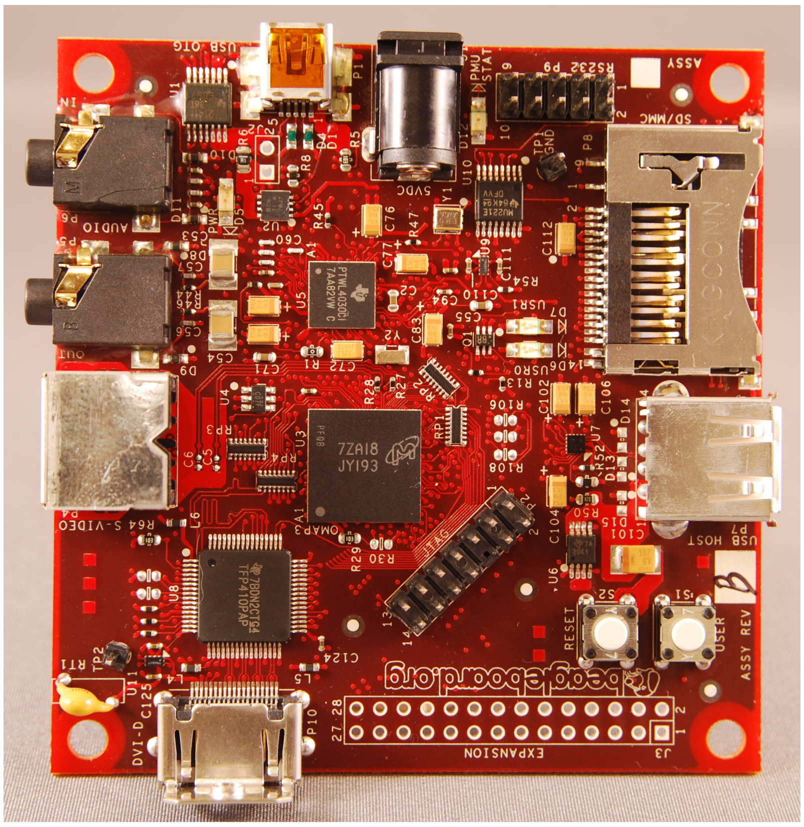


Figure 2.1: The BeagleBoard

Some key features of the BeagleBoard:

- OMAP3530 ARM Cortex A8 720 MHz CPU
- Integrated vector HW floating point unit
- 128 MB RAM, 256 MB flash memory
- ∼2 W power consumption

See [1, Skøien, Vermeer pp.13-17 & pp.19-29] for more information regarding the BeagleBoard, why it is used and available operating systems.

This system sets the guidelines for the software framework, though the solution is applicable in a much larger range of applications. Compared to an ordinary desktop system, this hardware puts no limitations on the developer, except for some reduction in processing power and no multi-core capabilities.

2.2 Framework Requirements

The software framework must accommodate the modularity of the hardware and provide a base for rapid prototyping and expandability. The following demands are specified by the author, based on the problem description, earlier work, experiences and requests from other UVL developers.

- Accommodate modules in several languages
  Modules in this setting are referred to as an independent unit which is either hardware

or software function specific. E.g. a GPS or GNC module respectively. MATLAB / Simulink support falls under this point.

- Highly modular design
  Adding a module should impose very little work in other existing modules.
  The user must be provided with a framework to allow rapid construction of new modules. The user should not be bothered with details regarding timing, inter-process communication (IPC) and data security/validity. Modules that use data from other modules must of course be aware of the structure of the posted data to make use of the information.

- Separate compilation
  To accommodate different languages and ease development, it is desirable to be able to compile modules individually.

- Possibility of different loop frequencies
  As an example, a typical GPS can update at 10 Hz, while an IMU can run at up to a 100 Hz. The framework must be able to allow for different execution rates, while having e.g. old GPS data accessible even though the IMU data is updated.

- Watchdog to monitor module performance
  A watchdog must make sure that each module updates its data at an interval given by the user. Should a module die, or provide data at a decreased rate, it can be rebooted without affecting other modules. (Of course except a prolonged interval of un-updated data).

- Maintain data integrity
  Care must be taken to ensure that corrupt data is not passed in the system, and that data is available for other modules while computation is ongoing in the posting module.

- The framework must be able to sustain soft real-time execution.

- Establish rules for new modules
  A software framework can indeed be generic, but to make any communication possible, new modules must be made in accordance with certain system-wide rules.

2.3 Previous Work

2.3.1 Former Implementation

In [1, Skøien Vermeer pp.34-36] a framework was suggested to enable different modules to communicate. A master thread spawned modules as threads and passed the necessary struct pointers. The use of classes were central throughout the solution. This system proved to work as intended, but had certain limitations:

- All the modules must be implemented in C++.
  This is one of the largest hurdles, since many developers might be unfamiliar with object oriented programming and the use of classes.

- Adding a module requires quite some modification of the central software block.
  The thread creation process is handled in the main loop. To insert a new module, one must spawn a new thread in `main` and pass a pointer to the appropriate struct.

One of the features that will be brought forward is the timing feature. As stated in [1, Skøien, Vermeer pp.34] this method is used which is originally based on [11, Simmonds]:

```
thread(period)
{
  struct periodic_info info;
  make_periodic (period,info);

  while(1)
  {
    /* Do work */
    wait_period (info);
  }
}
```

Where periodic_info is declared in the following manner:

```
struct periodic_info
{
  int timer_fd;
  unsigned long long wakeups_missed;
};
```

Before the while-loop is reached, a timing info struct is instantiated and a periodic time in μs is given as a parameter. In make_periodic, period is parsed to the correct format, and the command

```
info->timer_fd = timerfd_create (CLOCK_MONOTONIC, 0);
```

is run which creates a timer that via a file descriptor delivers timer expiration notifications. See [12, Linux Programmer's Manual]. When wait_period is called in the periodic loop, a blocking read call is made to the same timer:

```
ret = read (info->timer_fd, &missed, sizeof (missed));
  if (ret == -1)
  {
    //Handle error
  }
```

The scheduler will wake the threads at intervals to check the condition, which results in either going back to sleep or start executing a new iteration. How often this is done is dependent on many factors, such as type of scheduler, priority and system load. If the cycle time is too fast, the call will block and the thread scheduled out. On the other hand, if the computations in the loop are too large to be executed in that particular period or the program scheduled out for too long, the value of info->wakeups_missed will be incremented. This allows for both easy timing as well as a convenient way of knowing if modules miss their deadlines. The timing will not be accurate enough for hard real-time use due to its polling based nature, and the underlying non real-time OS. More about Linux real time properties can be found in [1, Skøien, Vermeer pp.27].

2.3.2 Some Comments on Other Possible Solutions Which Utilizes an OS

As stated, many UV projects before has utilized a simple microcontroller without an OS to control a craft. Such solutions have been examined to gather general information, but specific points will for the most part be of limited value in this setting where an OS is utilized.

What is much more interesting is how the software is structured in projects where an OS is in control of the system. One such project that is of great interest is *"Designing an autonomous helicopter testbed: from conception through implementation"* by Garcia [13, pp.61]. Here the idea of concurrently running processes communicating through shared memory structures is presented. The basic idea of using shared memory for inter process communication (IPC) originates from this dissertation, but the specific solutions commented upon later are chosen and created by the author. Further, the mentioned dissertation puts each module in responsibility of creating their own shared memory area, and the associated semaphore. This method will not be used here. In this project, to provide new developers with the simplest module framework possible, a centralized module handles the setup of shared memory and semaphores. With a central module being able to read data from slaves, it will have valuable monitoring capabilities. Another drawback is that in [13, Garcia] each module creates their own shared memory segment, and if that module fails before any other subscribes to it, the reserved memory will be released. If no processes are connected to the shared memory it will be marked for deletion by the OS. On the other hand, if both a master module and a slave connect to their shared memory, the slave can die whilst the memory is left available. Though no data is stored there, other modules will not fail trying to connect to this memory segment.

[14, Holmgaard et al. pp.23-30] also used shared memory between processes. They use a single large shared memory area protected by one semaphore for sharing variables. This implies some drawbacks in this setting such as: Every module must know the structure of the shared memory, even if all the data is not used by this particular module. If one wish to alter the memory size a module utilizes, all modules must know about this change. [14, Holmgaard et al.] keeps the structure of the shared memory is stored in an .h file, which means recompilation of all modules for a single change. This solution works well in more established projects, but here exists a desire for an even higher level of decoupling. If a module fails while holding the only semaphore the entire system will be affected, since no one can read or write to shared memory. This leaves the system highly dependent of each module functioning properly. In a continuously altering test environment as this thesis is targeted towards, it is desirable to be able to experiment with new modules. If a new module fails, this must impact the rest of the system to the smallest possible degree. [14, Holmgaard et al.] utilizes a single semaphore which will cause more waiting. With many modules accessing the same semaphore, the time spent waiting for access will increase as more modules are added and high module frequencies are used. Finally, with a single shared memory space it is possible to overwrite the reserved area. E.g. since many modules are using the same space, a developer writing buggy code might end up using more memory than originally planned, and thus overwriting what other modules have saved. In this thesis, a defined shared memory area for each module is reserved. This way the OS will ensure that a modules storage of data can not exceed a certain size.

2.4 Proposed Solution

2.4.1 Overview

In Figure 2.2, one can view the basic modular interconnection principle. Each module is responsible for posting data to its own shared memory segment, and make a local copy of the data from other shared memory segments of modules they subscribe to. The structure of this segment in the form of a struct is known to the other modules by an .h file which is available across the system, used only by the subscribing modules. Each module is as self-contained as possible, which means that each module spawns the appropriate thread(s), handles errors, and maintains control of execution rate. The basic principles for these actions are provided in the form of a framework.

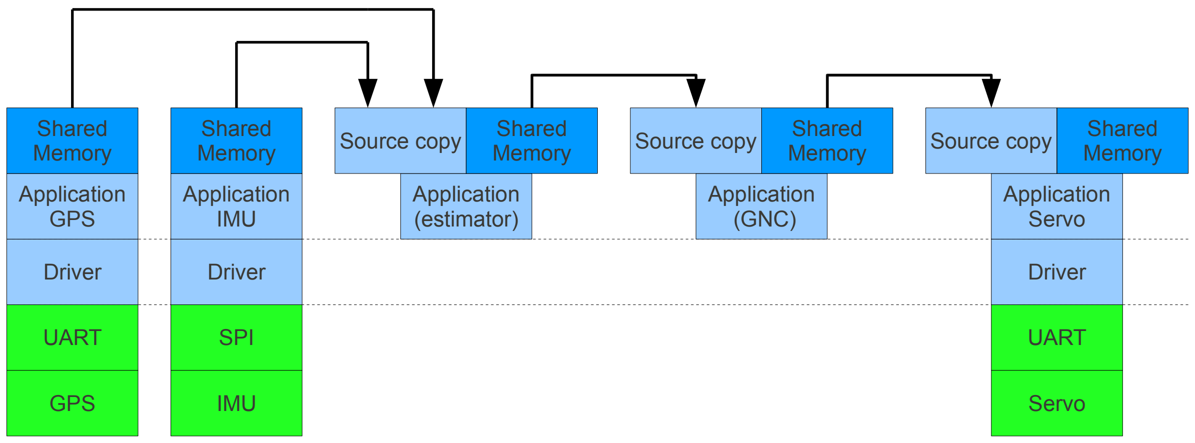


Figure 2.2: Software structure

The implementation is done in C++ but the framework itself is language independent. Languages that can make use of the system calls and POSIX commands in Table 2.1 can be used for an interface with the framework. A valuable added feature of this solution is the

System calls	
clock_gettime	shmat
freopen	shmctrl
popen	shmdt
sem_close	shmget
sem_open	signal
sem_post	
sem_unlink	
sem_wait	

Table 2.1: Utilized calls

possibility to incorporate MATLAB/Simulink. Via S-functions and Real-Time Workshop, the control system can be developed in Simulink, which is a familiar environment for many engineers today in the academic and industrial sector. Sensors appear as blocks and the calculations for servo/actuator settings is placed in shared memory for the servo control module to subscribe to. Such an interface is discussed in [1, Skøien, Vermeer pp.35-40]. Figure 2.3 depicts a simplified image of how the system might look in Simulink once all the modules are finished. The left side boxes reads from shared memory and the expansion board box on the right posts to shared memory. All the functionality of the GNC block is developed in Simulink.

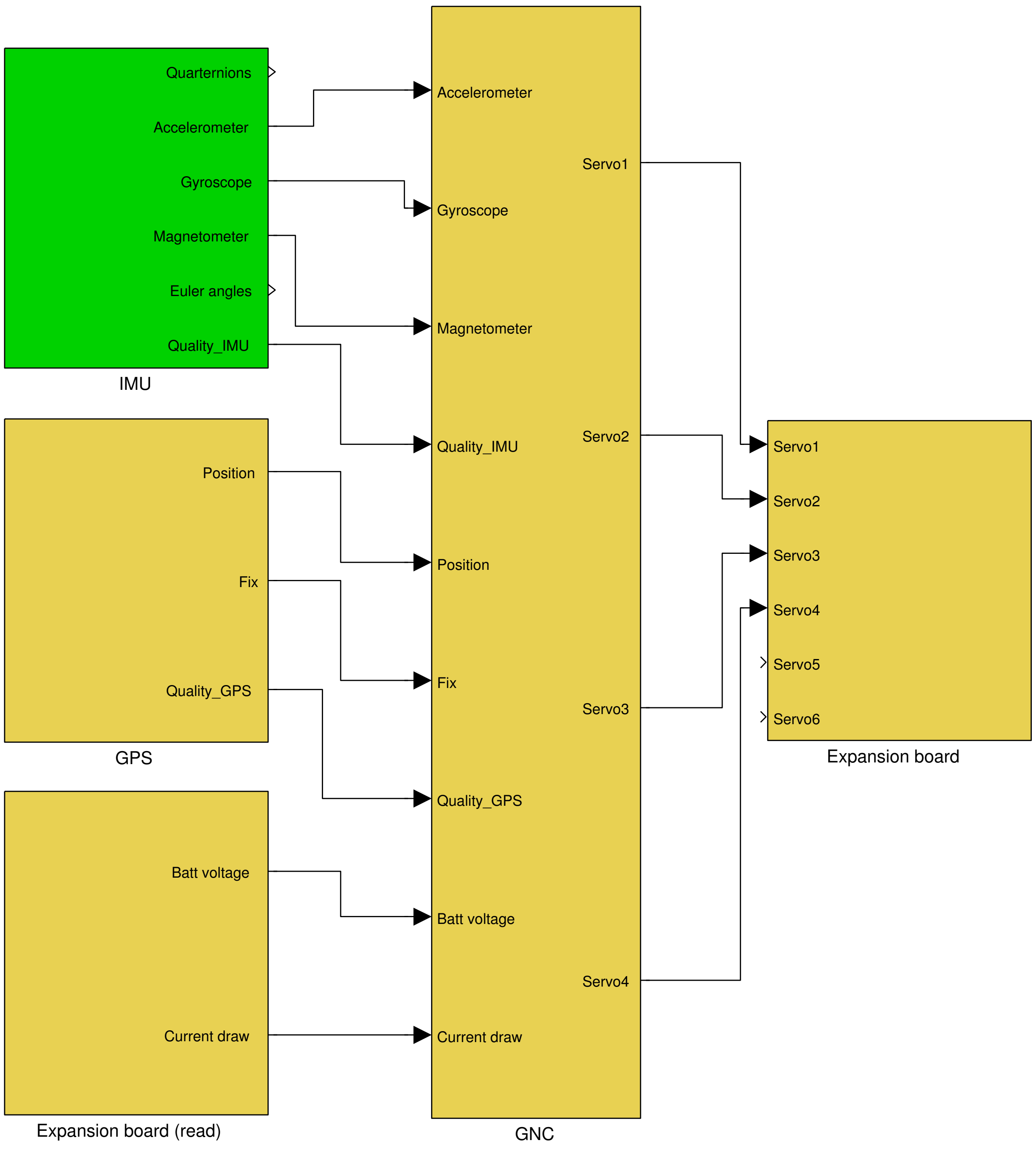


Figure 2.3: Simulink example

The box marked green is finished and the yellow are yet to be completed. The IMU and Simulink interface is discussed in Chapter 3 and the expansion board in Chapter 4. It is emphasized that utilizing Simulink it entirely optional.

2.4.2 Shared Memory

As mentioned, shared memory will be used as the primary means of communication between the modules. This solution has the advantage of being a very fast way of transferring data, keeping information available regardless of other work being performed and is accessible throughout the system. As seen in Figure 2.2, each module is responsible of posting data to its shared memory segment at the given intervals of the module. Data posted to shared is in the form of a struct of a given size that is structured similar to this:

```
struct struct_dummy1{
  long timestamp1_ms;                 // Last timestamp from module
  long max_time_ms;                   // The maximum allowable cycle ↩
     time
  unsigned long long missed_deadlines; // Missed wakeup calls
  long timestamp2_ms;                 // Last timestamp copy
  char name_general[50];              // Module name
  char name_specific[50];             // Additional information
  int mem_key;                        // Shared memory key
  int pid;                            // Process PID
  int data_1;                         // Data
  int data_2;
  .
  .
};
```

Listing 2.1: A typical struct residing in shared memory

The master module needs the timing information to determine if the module functions as intended. The first four variables are used for this purpose. These must always be located at the top of the struct, since these are cast to a timing structure in the master module. This part of the structure is fixed so the master need not know the remaining fields. `long timestamp1_ms` consists of UNIX time plus three digits which are the milliseconds. `long max_time_ms` is checked by the watchdog against `long timestamp_ms` to determine if the module has timed out. `unsigned long long missed_deadlines` is written to by the timing feature. If the module misses wakeup calls, it means it does not have enough time allocated by the system. Increasing the cycle time, increasing the priority or reducing the system load can correct this issue. `long timestamp2_ms` is used by the master to verify data validity, and should hold the same value as the first timestamp. The master can not use semaphores, since there is a risk of being blocked by a slave malfunctioning whilst holding the semaphore, and thus not be able to read the timing data. Since the master module does not use semaphores when reading shared memory, it needs some method of data verification. If timestamp 1 & 2 are equal, the master can verify that the data is valid. As an example, a slave might lock the semaphore and start writing to shared memory. As this happens, the slave is scheduled out, and the master starts reading the same memory, which can happen when not using semaphores. This would yield two different timestamps, and thus invalid data.

The rest of the variables are up to the maker of the module to decide upon.

Figure 2.4 gives another view of the shared memory solution. The system master has access to all the shared memory segments, but does not need to know information about the structure (residing in the .h files) due to the mentioned fixed data fields.

Modules that only post data to the system (e.g. GPS) only have shared memory to post its data, and does not subscribe and make source copies. On the other hand, modules that read/subscribes to others need to know the structure and thus have access to the .h files of the supplying module. To minimize access to shared memory, each subscribing process makes a copy of its provider rather than work directly off the shared memory data. Directly working off the shared memory segment would cause the semaphore to be locked a great portion of the time. (More about semaphores in 2.4.3). Modules that only subscribe to others such as the Application logger uses its shared memory only to inform the master of their status. Since there are no more specific data fields in shared memory, there is no need for an .h file to describe the structure. Although not shown in Figure 2.4, each shared memory module has a semaphore associated with it.

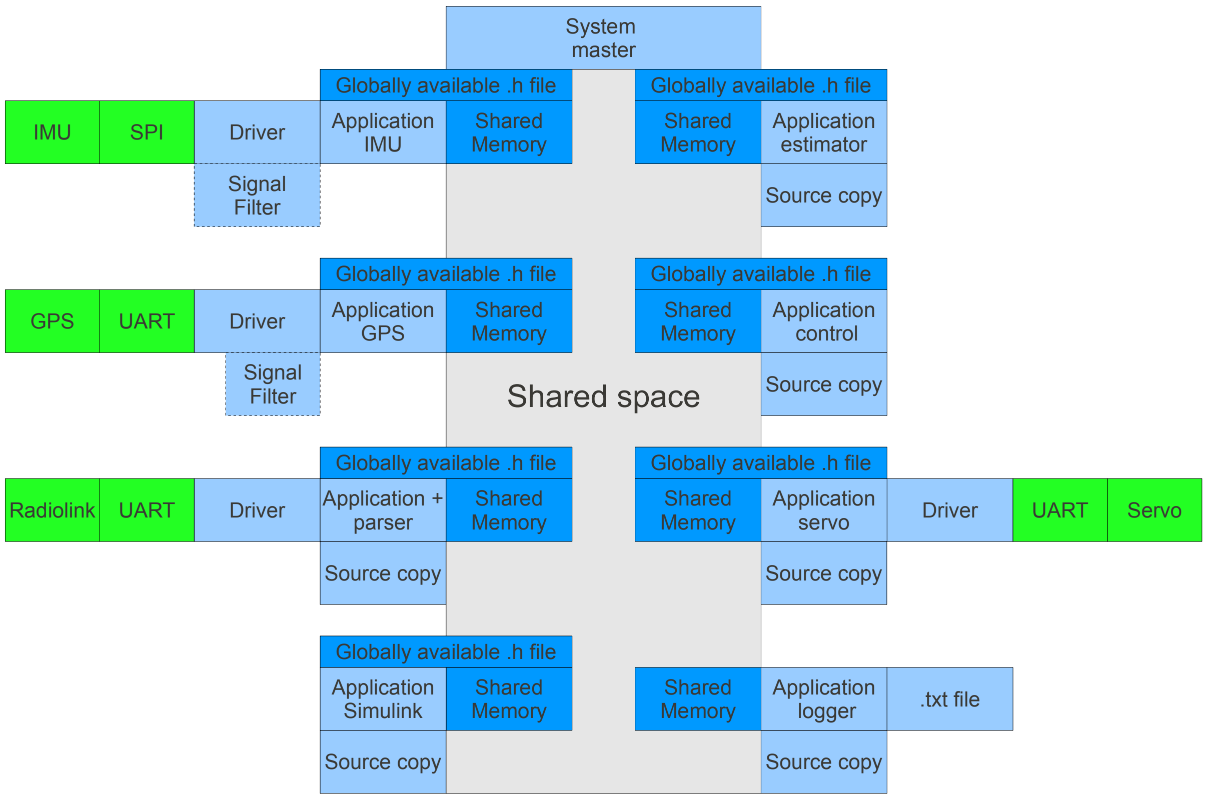


Figure 2.4: Shared space

2.4.3 Use of Semaphores

Semaphores are in the most basic form a mechanism that can enforce mutual exclusion and synchronization through coordination of access. (Semaphores will not be commented upon in depth. For more information, see [15, Burns and Wellings pp.233-235]). In this software framework, named binary semaphores are used for both these functions. The primary function is to provide mutual exclusion to the shared memory regions. Consider the following scenario: The owner of a shared memory region starts to write data but is preempted by the scheduler during the write operation. If a reader is then scheduled to run, it can now access the memory region only to read data that are only partially updated or even corrupted. Semaphores are also used to hold the execution of all the slaves until the master has finished its initialization.

Named semaphores are accessible through the entire system as long as the same identifier is used. The filepath of the modules are used to generate the name, so each module knows the name of its provider's semaphores.

Two main commands are used during program execution to use the semaphores:
`sem_wait(semaphore)` Decrease the value of the semaphore from 1 to 0 and continue. If the value is already zero it means the semaphore is locked and the process must wait. Execution is continued in another thread or program.
`sem_post(semaphore)` Increases the value from 0 to 1 to release the semaphore. A snippet from the program where a slave writes to shared memory:

```
sem_wait(semaphore);
*struct_ptr = *struct_ptr_local;
sem_post(semaphore);
```

Note that all updated data is placed in `*struct_ptr_local` before all the data fields in the local struct are copied out as a whole to minimize the locked time of the semaphore.

2.4.4 Startup Sequence

In some systems, the startup sequence is extra crucial due to the fact that each module creates their own shared memory segment. (Such as in [13, Garcia]). Modules that use data from other shared memory sources are dependent of these areas being created beforehand. By centralizing the initialization of shared memory in the master, it is only a matter of holding the execution of all slaves until all the shared memory is allocated.

In the following figures, the functionality is illustrated with two slave processes, -one data subscriber (SLAVE1, ”dummy1”) and one data provider (SLAVE2, “dummy2”).

Consider Figure 2.5 which illustrates the following explanation of the startup sequence:

The system first receives a start command from the user or the system. The master module begins traversing through the file paths of the slaves to be started. In the `system_master.h` file resides the description of all the slaves which are provided to the master in the following manner:

```
string modules[] = {"sudo nice -n -5 ./dummy1/dummy1","sudo ./dummy2↩
   /dummy2"};
```

All the executables are listed in an array, where optional priorities can be given. Here the first slave (dummy1) is executed with a niceness of -5 with respect to the systems default priority setting, implying that it has a higher priority than dummy2. In Linux, priorities range from -20 (highest) to 19 (lowest). Although not implemented, the command `renice` can be used to increase or decrease the priority of modules at runtime.

Firstly, the last 4 letters of SLAVE1 (“dummy1“) are used to generate a numerical key which are used for identifying the shared memory region. One will be alerted if the name is too short or duplicate names are found. The string for the semaphore identifier is the same as this numerical key, making it a simple matter to associate shared memory and the belonging semaphore. Semaphore 1 is created in a locked state. SLAVE1 is started with the command

```
pf = popen(arguments.c_str(),"r");
```

where popen starts a process connected by a unidirectional pipe which is read ''r'' back to the master. arguments is a string on the form sudo nice -n -5 ./dummy1/dummy1 1835890993 &. The shared memory key could of course be generated by the slave itself, but is transferred as an argument to save code. The slave in turn transmits back its PID and size needed in shared memory via a pipe and reroutes the pipe to the terminal to output potential messages:

```
freopen("/dev/tty", "w", stdout); // reroute the pipe to terminal
```

A sem_wait then holds SLAVE1 from further execution. The master creates the shared memory region for SLAVE1.

The same sequence is repeated for all slaves before the master releases all semaphore locks and the slaves begin executing. They connect to their own shared memory regions, and to other slaves shared memory that they might subscribe to based on their names. This way the master assures that all modules have started without errors, semaphores are created and locked, and that all memory has been successfully allocated before commencing execution.

2.4.5 Normal Operation

Consider Figure 2.6. During normal operation, each module executes at its defined frequency which usually is in the range of 1-100 Hz. This execution rate is defined beforehand by the user, but can also be altered during runtime by changing the make_periodic (period,info);. Some modules might utilize blocking calls to the system or external units, and these can be used to make the module execution periodic. One should be careful not to use busy-while implementations that will use system resources unnecessarily. In this example, SLAVE2 writes data to its shared memory at given intervals. At one point in time, SLAVE1 reads the data from shmem 2 to process that data. SLAVE1 posts its processed data to shmem1 for other modules to utilize.

At given intervals, the master unit reads the timing data from all the slaves to ensure that they execute periodically via the watchdog. A reboot (SIGQUIT signal sent and then handled) can be performed on non-functioning modules if desired and possible. See Section 2.4.7. Note how the master reads data from SLAVE1 with the semaphore in locked state as mentioned earlier.

2.4.6 System Shutdown

Consider Figure 2.7. When a SIGINT is caught by the master module, it iterates through the slaves sending SIGTERM. Notice that SIGTERM is used for signaling shutdown whilst SIGQUIT is used for signaling slave reboot, enabling slaves to take different actions. Each slave stops its current execution, takes appropriate shutdown actions/ precautions and exits. The master module deletes all the shared memory and finally closes all the semaphores.

2.4.7 Watchdog

The watchdog is an optional service provided to the system modules which resides in the master module. It is able to send signals to processes that do not respond (post shared memory data) within a given time, and restart the module if desired and made possible by the developer. Each module supplies a timestamp of the current system time when posting

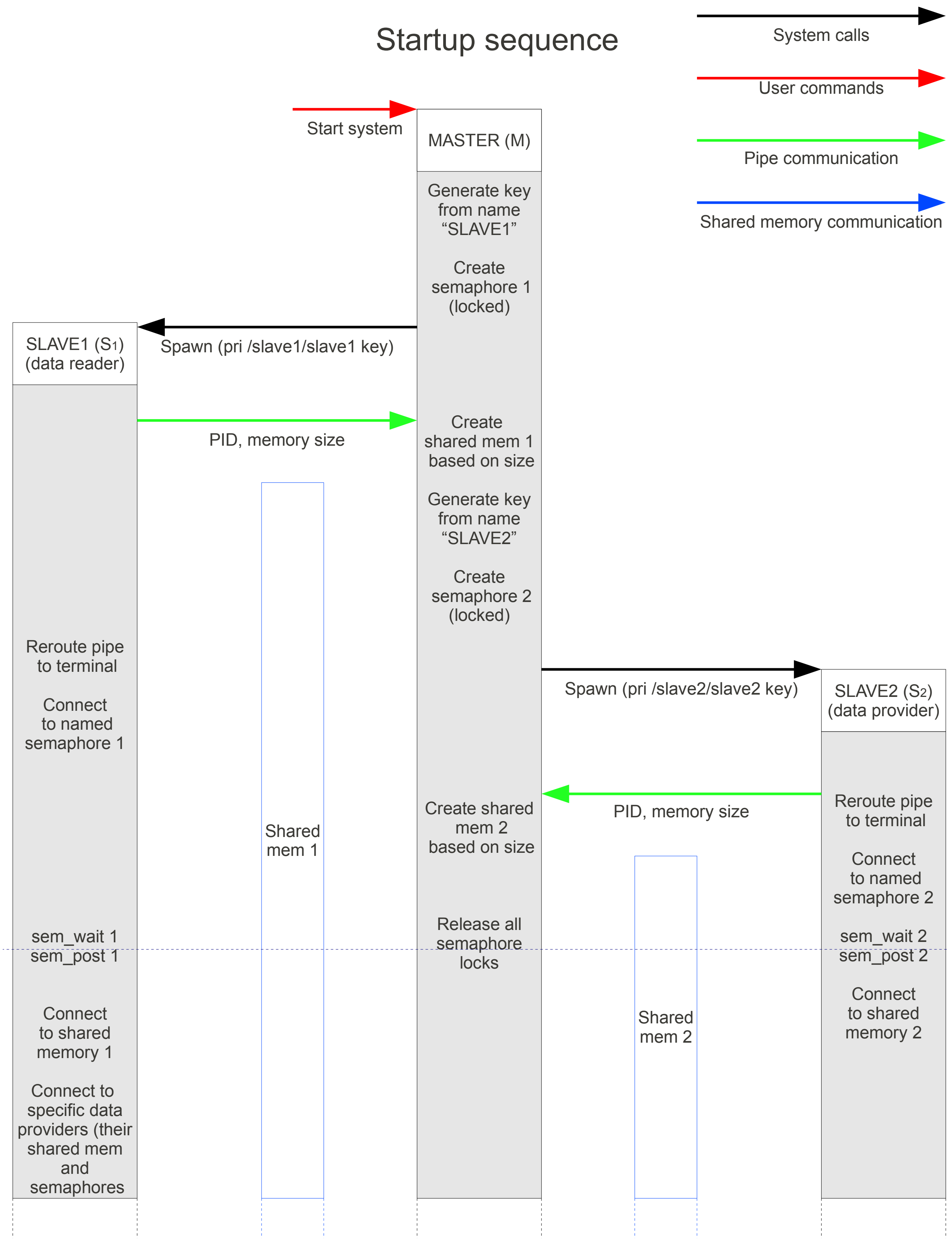

Startup sequence
System calls
User commands
Pipe communication
Shared memory communication
Start system
MASTER (M)
Generate key from name "SLAVE1"
Create semaphore 1 (locked)
Spawn (pri /slave1/slave1 key)
SLAVE1 (S1) (data reader)
PID, memory size
Create shared mem 1 based on size
Generate key from name "SLAVE2"
Create semaphore 2 (locked)
Reroute pipe to terminal
Connect to named semaphore 1
Spawn (pri /slave2/slave2 key)
SLAVE2 (S2) (data provider)
Shared mem 1
Create shared mem 2 based on size
PID, memory size
Reroute pipe to terminal
Connect to named semaphore 2
sem_wait 1
sem_post 1
Release all semaphore locks
sem_wait 2
sem_post 2
Connect to shared memory 1
Shared mem 2
Connect to shared memory 2
Connect to specific data providers (their shared mem and semaphores

Figure 2.5: Startup sequence

the data to shared memory, a maximum allowable delay and its missed deadlines. This is done in the `long timestamp_ms1`, & 2, `long max_time_ms` and `unsigned long long missed_deadlines` variables respectively.

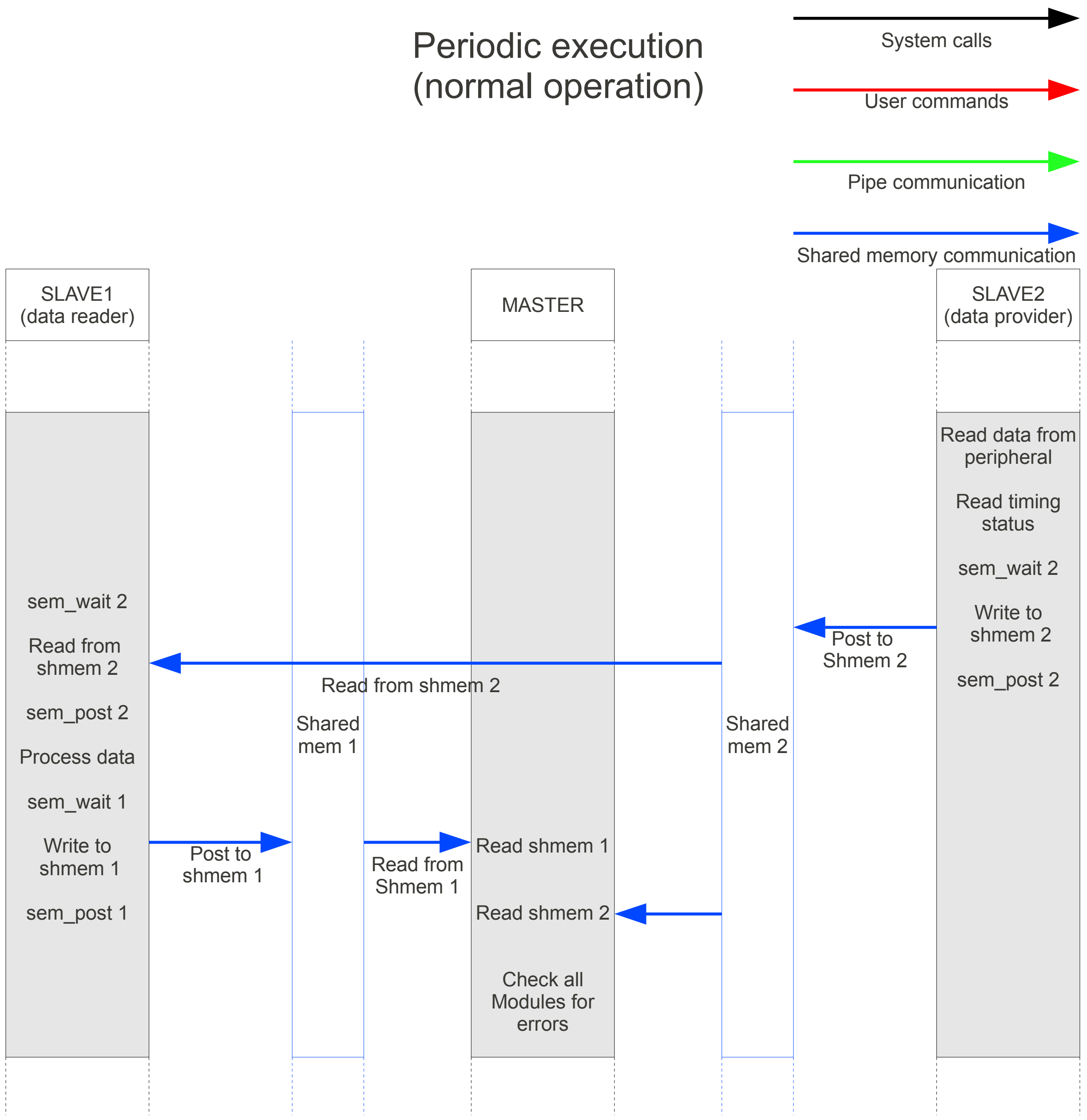


Figure 2.6: Normal operation

Figure 2.8 shows a simplified flow scheme. The watchdog is initialized by connecting to each modules shared memory. WATCHDOG_START_DELAY is defined to keep the watchdog from potentially rebooting modules that need time to start, before they can post updated timing info. The watchdog traverses all the slaves each cycle of the master module, copying out the timing information from shared memory to local storage. It begins by examining the `long max_time_ms` value. Currently the configuration is such that a -1 here equals no watchdog active for this module. Should this field contain a positive value, it signifies the max allowable time in ms without response before the module should be considered as faulty. With the current configuration, the watchdog sends SIGQUIT to the given module and then reboots it. The developer can choose what actions should be taken when SIGQUIT is caught.

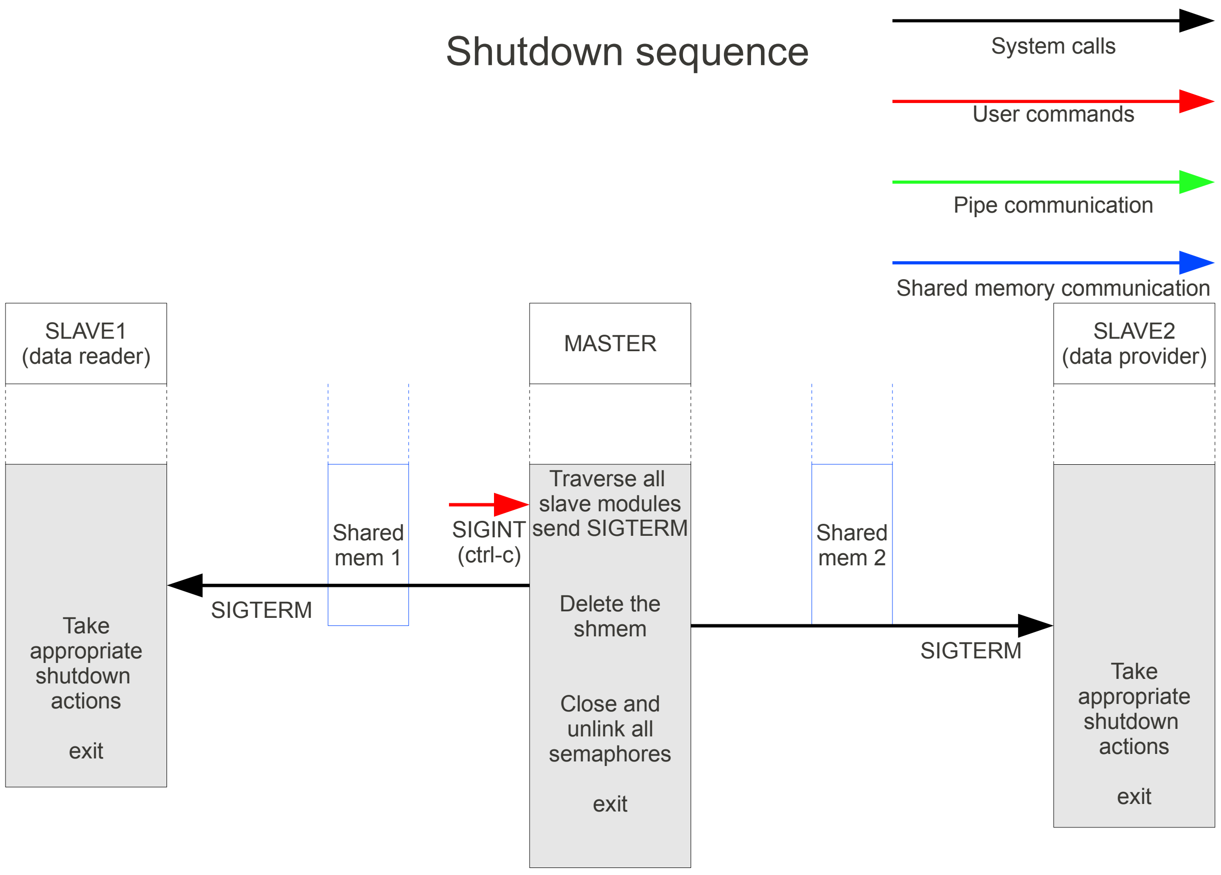


Figure 2.7: System shutdown

2.5 Tests and Results

2.5.1 BeagleBoard Test

To verify basic functionality, the software framework was cross-compiled and tested on target. A detailed guide to recreate this test is found in appendix B. The master was set to run at 1 Hz, dummy1 (D1) reads an integer from shared memory at 1 Hz and prints the value to terminal. Dummy2 (D2) increments the same integer each cycle and runs at 100 Hz. The frequencies are chosen to be in the vicinity of what the system might experience in a UV application.

The master was started with the command:

```
root@beagleboard:/home/cross_comp_test ./system_master
```

The master begins by spawning the two slaves:

```
MASTER started module: ./dummy1/dummy1 pid: 1038
MASTER started module: ./dummy2/dummy2 pid: 1040
```

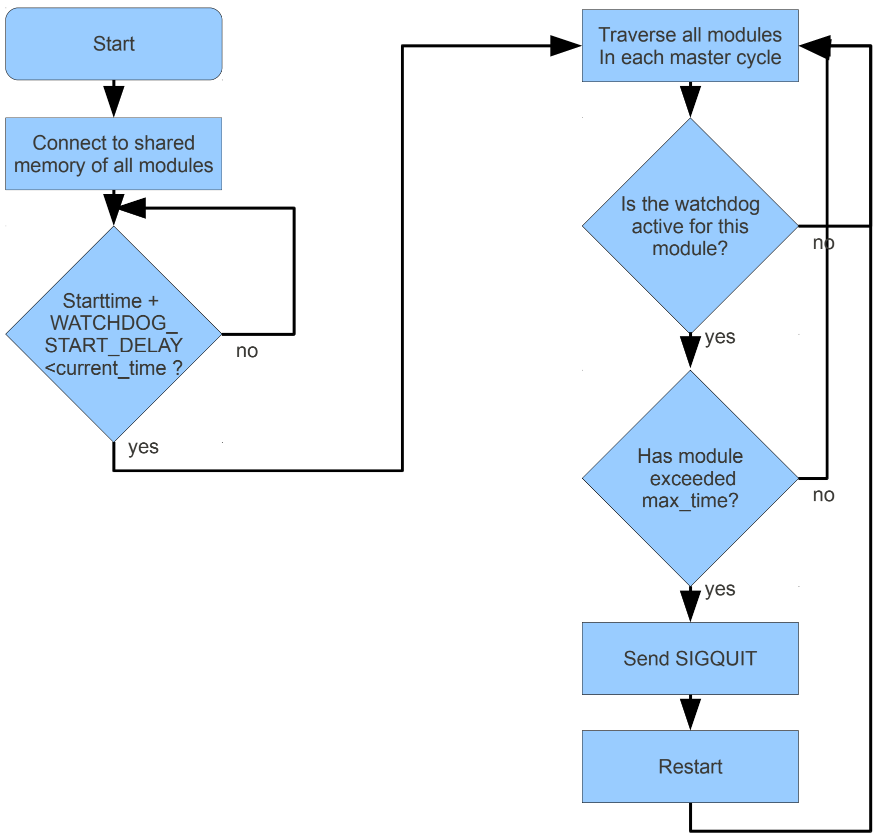

Start
Connect to shared memory of all modules
Starttime + WATCHDOG_ START_DELAY <current_time ?
no
yes
Traverse all modules In each master cycle
Is the watchdog active for this module?
no
yes
Has module exceeded max_time?
no
yes
Send SIGQUIT
Restart

Figure 2.8: Watchdog flow scheme

Each time the master runs, it prints status information from all the slaves. Here is an example from this test where 0 and 1 denotes dummy1 and dummy2:

```
MASTER in watchdog module: timestamp1 : 0 643594065
MASTER in watchdog module: max_time  : 0 10000
MASTER in watchdog module: missed deadl: 0 0
MASTER in watchdog module: timestamp1 : 1 643594311
MASTER in watchdog module: max_time  : 1 10000
MASTER in watchdog module: missed deadl: 1 0
```

The timestamps are on the format stated earlier, the watchdog timeout is set to 10 seconds for both slaves, and missed deadlines are reported.

To get an estimate on missed deadlines (MD) as a function of module frequency, the tests in Table 2.2 were run. More tests have been performed in the area where missed deadlines start to appear. Unrealistically high execution rates are also run to examine how missed deadlines scale with regards to module frequency. The one minute tests were run three times to get a better average estimate on the number of missed deadlines. However, this average has a very large standard deviation, due to substantial differences in the amount of missed deadlines for each run. A rough estimate of CPU usage was obtained via the "top" command. The `nanosleep` command was also tested to examine if this yielded fewer missed deadlines, but the results were far worse. Results are discussed in 2.6.1.

Test time	D1 exec	D2 exec	D1 CPU	D2 CPU	D1 MD	D2 MD
1 min	1 (1 Hz)	10 ms (100 Hz)	∼0	∼0.3%	0	0
5 min	1 (1 Hz)	10 ms (100 Hz)	∼0	∼0.3%	0	0
10 min	1 (1 Hz)	10 ms (100 Hz)	∼0	∼0.3%	0	0
30 min	1 (1 Hz)	10 ms (100 Hz)	∼0	∼0.3%	0	0
60 min	1 (1 Hz)	10 ms (100 Hz)	∼0	∼0.3%	0	0
12 hours	1 (1 Hz)	10 ms (100 Hz)	∼0	∼0.3%	0	0
1 min	1 (1 Hz)	2.5 ms(400 Hz)	∼0	∼3%	0	0
5 min	1 (1 Hz)	2.5 ms (400 Hz)	∼0	∼3%	0	13
10 min	1 (1 Hz)	2.5 ms (400 Hz)	∼0	∼3%	0	14
1 min	1 (1 Hz)	1 ms (1000 Hz)	∼0	∼8.4%	0	0
5 min	1 (1 Hz)	1 ms (1000 Hz)	∼0	∼8.4%	0	20
1 min	1 (1 Hz)	0.85 ms (∼1176 Hz)	∼0	∼9.8%	0	7
1 min	1 (1 Hz)	0.75 ms (∼1333 Hz)	∼0	∼10%	0	19
1 min	1 (1 Hz)	0.5 ms (2000 Hz)	∼0	∼17.5%	0	186
1 min	1 (1 Hz)	0.25 ms (4000 Hz)	∼0	∼35.5%	0	1123
1 min	1 (1 Hz)	0.1 ms (10 kHz)	∼0	∼36%	0	∼300000

Table 2.2: Framework test scenarios

2.5.2 Shared Memory Access

A rugged test was run on the BeagleBoard to examine the overhead caused by shared memory access in the framework. To represent typical execution rates, tests are run with a single slave, executing at 0.1, 1, 5, 10, 50 and 100 Hz, and the master module with watchdog running at 1 Hz. The slave does no work, except for posting its execution data to shared memory in the same format as in Listing 2.1. The framework overhead percentage vs. slave frequency is presented in Figure 2.9.

The process of getting the timestamp, unlocking, posting to shared memory and locking the semaphore, was run thousands of iterations to get the average time. It was found that in this test case, the mentioned process took appropriately 6.28 μs every slave cycle, which yields 0.00063% overhead at 1 Hz and 0.063% at 100 Hz. This number is not very accurate, but it does not need to be to get an estimate of the overhead. The accuracy is commented upon in 2.6.2. This overhead percentage will scale almost linear as the execution rate increases, but the time used for posting remains constant.

2.6 Discussion

2.6.1 BeagleBoard Test

As one can observe from the test data in Table 2.2, missed deadlines become apparent around 400 Hz. This number has limited practical significance, as the modules tested here does almost no work, and there will most likely be far more modules in a typical application. The variance in missed deadlines is very high. The five minute test at 400 Hz revealed missed deadlines in the range of 0-50 for each run. By observing the missed deadlines over time, an interesting discovery was made. The system can execute for minutes at the time

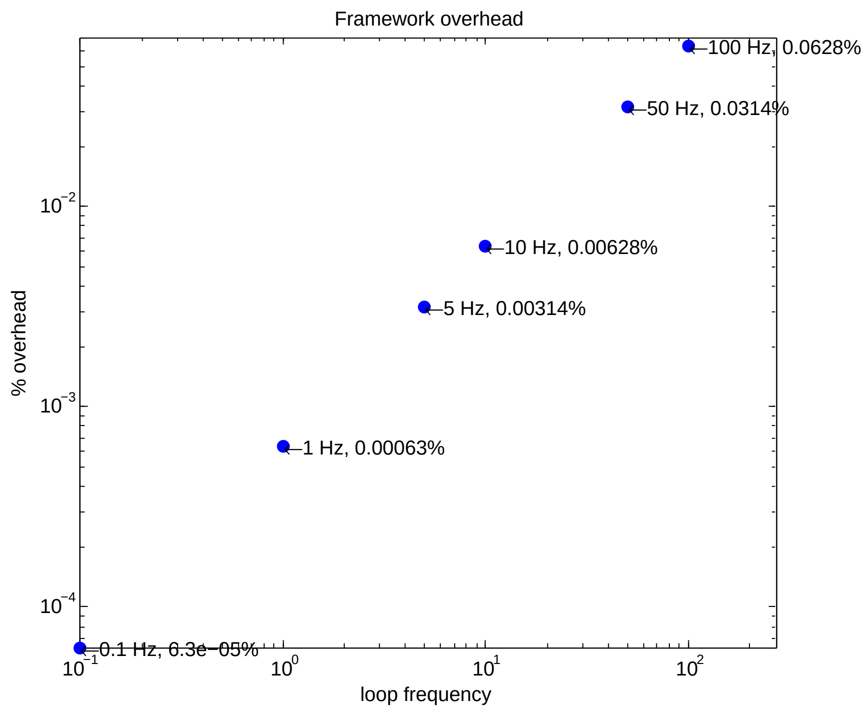


Figure 2.9: Framework overhead under test conditions

without losing deadlines, when suddenly 40 can be missed in the matter of seconds, and then again resuming execution without losing further deadlines. This can imply that the OS does some other work at times, which will severely impact the framework if it operates at the edge of the OS's capability.

Another interesting result is that the system looses deadlines even if CPU usage is low. This has to do with the overhead from context switching. This mechanism which is used to share threads over a processor needs time to store the program counter, registers and so on each time a switch is performed. The higher the switch rate, the more time is used switching back and forth between processes without doing any actual work from a users point of view.

In a normal (Vanilla) kernel, this overhead can contribute significantly, as can be observed in [16, David et al.]. With more code in the modules, this CPU to missed deadline ratio will improve, since more CPU is used compared to the number of context switches. This result also suggests that very high frequency modules are not ideal, especially on this OS, since the number of missed deadlines scales extremely fast with the number of context switches. The framework should be tested on a real time OS, or at least with a real-time scheduler that improves the task switch overhead to examine the performance.

To summarize, one can not guarantee that the system meets all its deadlines with the current timing system and non real-time OS.

A vast number of scenarios could have been run with parameters such as: number of slaves, execution rates, the amount of work done in slaves, other applications on the system, different priorities and more. This would have been too comprehensive in the timeframe of this thesis. The test is done to verify functionality, and show that deadlines will be missed if system load is too high.

2.6.2 Shared Memory Access Overhead

This test was run to get an estimate of the overhead associated with shared memory access, and to examine how this access impacts the system performance in normal and extreme cases. Due to the fact that the shared memory access time is very short, the use of normal timing features to examine it is rendered useless due to the high granularity. Time was spent examining how to access CPU tics to achieve the desired accuracy, but no solutions were found that could be easily performed. This lead to the use of the less accurate `clock_gettime`, with the memory access run thousands of times in succession and this again run 20 times to get a descent mean value. One must be aware that the system load, size of shared memory and scheduler plays a crucial part in the shared memory access time. If the process is scheduled out while holding the semaphore due to Linux operations or other programs, that particular access will take somewhat longer. There is thus some variance in the access time, but without some low level timing mechanism this variance will remain unknown. Even so, the test determines that the access times are generally very short compared to the cycle time, so that in all practical cases, the shared memory access time does not influence the overall performance of the system in a significant way.

2.6.3 Goodness of the Solution

The benefits and drawbacks of the solution are summarized here and commented upon.

By using separate shared memory regions, several advantages are gained. Firstly, the time spent waiting is less compared to if all modules were to access a large common shared area each cycle. Secondly, this solution increases security as no module can overwrite a larger area of memory than what was originally reserved. Finally, if a module fails while holding the semaphore, only the subscribing modules will be affected. This implies a prolonged interval without availability of new data. Subscribers can use `sem_timedwait` when reading from others. This allows for waiting a certain amount of time, and if the semaphore is not acquired by then, the module continues its execution. By doing this, subscribing modules will not freeze for an infinite amount of time if the semaphore is unavailable.

Due to the fact that there is a very limited chance of a module failing in the short amount of time it is holding the semaphore simply to copy out data, `sem_timedwait` is not standard in the framework. It has though been successfully tested, and it is up to the developer to choose if a module must acquire data each cycle, or can suffice with older data for some time.

By having the master set up all the shared regions before releasing the slaves ensures that there is enough allocatable memory, and that all regions have been set up correctly.

All modules copy in and out the entire shared structure every time, working off the local copy, which minimizes the locking and unlocking of semaphores.

By enforcing some fixed fields in the memory structure, the master module and watchdog can keep track of module execution and take necessary actions without knowing the total structure. Such a method makes for a highly decoupled system where changes in memory structure are of no concern to the master. Only the subscribing modules need to know of this change to make use of the structure in shared memory.

As of now, all the modules scheduled to start are stored in the `system_master.h` file. This implies that all modules must be present in the header and compiled with the master.

The contents of the header file are read at boot only, which means that dynamic exchange of modules is not implemented but indeed supported by the framework. One must be extremely vigilant if such a feature is implemented. - The master and many subscribers might be connected to the module one wishes to change. This means that if e.g. the module "slave2" is exchanged with "Xslave2", they must have the exact same shared memory structure to not make the data invalid for subscribers, they must connect to and post to the same shared memory region, and one must ensure that slave2 is completely killed before Xslave2 is spawned so two modules do not post different interleaved data.

One can discuss the value of such a high risk feature, compared to sending a signal to a module, and changing e.g. to another test algorithm internally. It is possible to manually change a module, by killing "slave2" and replacing the executable with another slave with the same name. The watchdog will detect the timeout, try to kill the non-existent module and restart the new module. This is not recommended as this sequence is prone to user errors and is far more complex.

The framework is ready for use, and developers are presented with a dummy, which they can modify quite easily to fit their needs. A small drawback is that there is quite an amount of code in these files already to enable interaction with the framework. However, new modules can be constructed fast, interfaced quickly and tested without giving much thought about other modules in the system. The system allows for rapid prototyping, easy expandability, use of different languages, confinement in modules, flexibility, complements hardware modularity, error detection and error correction.

The system can not guarantee any hard real time demands, as this is heavily dependent on the underlying OS. As discussed in [1, Skøien, Vermeer pp.20], hard real time is not usually necessary for control in the unmanned vehicle domain. The operator will however be alerted if modules miss their deadline, and tests run so far have yielded good results.

Chapter 3

Sensor Signal Quality and Error Detection

3.1 Background

In every possible genre of automation, sensor equipment is used to provide information about the state of the system to the feedback loops, enabling automatic control.

All sensors have a chance of failing and thus transmitting an erroneous signal or no signal to the control system. Undetected signal errors can be fatal for the control loop as incorrect feedback is given, and the wrong or inadequate action is performed. Consider a constant signal from an aircraft altimeter which will cause the autopilot to believe the current thrust (and elevator setting) is sufficient to maintain the altitude. Fuel saving algorithms programmed into the system will try to reduce thrust by as much as possible, whilst maintaining altitude. The result might be fatal. This scenario is highly unlikely on commercial aircraft due to human interaction, redundancy and smart algorithms. However on a simpler UAV this scenario might occur.

Signal error detection is rarely a part of the guidance, navigation and control (GNC) system itself. It is often implemented near the HW, and is very sensor specific. Should an error occur, a message must be sent to the control system which can choose to use degraded data if possible, discard the sensor data or make another appropriate action. A framework applicable in the unmanned vehicle domain should be equipped with such functionality.

3.2 Scope

A broad range of sensor equipment is utilized in unmanned aerial vehicles. Gyroscopes, magnetometers, accelerometers, GPS, barometer, ultrasonic and pitot sensors are just a few examples. Quite often the first three mentioned sensors are integrated into one component and named an inertial measurement unit (IMU), giving the attitude of the vehicle.

As the unmanned aerial vehicle project progresses, the need to have a form of low level signal quality detection and alert system in the framework is present. A sensor signal quality detection algorithm is constructed, which is very general to accommodate a broad range of sensors. Among other HW, the UAV's IMU is run through this module, and algorithms created to detect and alert the autopilot and operator when IMU data is erroneous.

The detection system within this thesis provides basic and general error detection. Extensive theory exists on the subject, but much is beyond the scope of this initial implementation, such as use of multiple IMUs which are uncommon on smaller unmanned systems, and larger models to accommodate such parameters as scale factor and nonorthogonality. Some solutions rely on knowing the actuator input to see if the sensors respond accordingly to find errors. See [17, Heredia et al. pp.96].

The implementation is targeted towards general analog sensors, and for illustrational and development purposes, the Xsens IMU is used here. Other sensors such as pitot or ultrasonic height sensors can also with ease be monitored as many errors such as high noise, drift or no-signal can occur in many measurement devices.

3.3 Theory

3.3.1 Main Error Sources

Flenniken IV et al. [18, pp.1] identify three main accelerometer and gyro error sources:

- Constant offset (bias)
- Moving bias (sensor drift)
- Random error

Vik [19, pp.37] also point out three main IMU error sources:

- Bias
- Scale factor
- Misalignment (sensor placement with regards to vehicle frame)

Some of these points have to be considered in the design and building stages. More crucial errors that can appear during flight that requires immediate attention is the main focus of the detection algorithm. Rogers [20, pp.4267] points out some more general faults in analog sensors which are more in-line with the target in this project:

- Dead
- Excessive noise
- Drift
- Offset (bias)

Rogers[20] points out the fact that the drift and bias require specific models to be corrected. The sensor quality algorithm is supposed to be general and applicable to almost any signal. In addition, many sensors have built-in bias correction. As an example, the Analog Devices ADIS16405MLZ and the Xsens MTi IMU have such functionality. Finally, the observers are better suited at detecting drift and offset if necessary. In contrast to the general sensor algorithm, an observer has access to measurements from many sensors such as GPS, IMU and pitot. Since all these measurements are interconnected, it is possible to compare measurements from several sensors to detect errors. To draw a conclusion, the low level general error detection must be able to detect the most crucial low-level errors whilst being applicable to most sensor outputs.

Based on the previous documentation, [21, Sharma et al. pp.3], and input from others, the detection system is targeted towards:

- Excessive noise
- Dead sensor
- Frozen output
- Short noise spikes
- Unrealistically high and low values

How to find these errors are presented in Section 3.5.

3.3.2 Definitions

Before the solution is embarked upon, some appropriate fault terminology is defined. Consider Figure 3.1 as a reference for the following examples where an oscillating device is mounted to the IMU to simulate fuselage vibration. The unfiltered accelerometer in the Z (down) direction is presented here. Mind that although the data is from a physical IMU, the errors presented are artificial for purpose of illustration. [21, Sharma et al. pp.5] identifies three main signal faults:
NOISE (Figure 3.2): The variance of a window of samples are higher than a given threshold. This can be characterized as many SHORT faults in succession. Unlike the SHORT faults, the NOISE can be due to e.g. a newly introduced source of noise in the system or a faulty sensor transmitting random values.
CONSTANT (Figure 3.3): A given sensor transmits the same or very similar values over a range of samples. This can in imply a frozen or dead sensor.
SHORT (Figure 3.4): A sudden change in output value compared to the previous sample. This could indicate a short physical jolt or corrupted data at one point in time.
In addition, signals that are unrealistically high or low for their domain are addressed.
HIGH: Values above a certain threshold.
LOW: Values below a certain threshold.

3.4 IMU Interface

3.4.1 Background

To develop and test the error detection system, an IMU will be used to provide example data to the system. Since it has been interfaced before and is known to provide high quality data, the Xsens MTi IMU is used for this purpose. Due to the cost of this device which is in the region of 1750 EUR (fall 2010), this IMU is not suited for the final product due to the focus on low cost. The MTi is shown in Figure 3.5 and extensive information is available from the Xsens homepage [22]. A GPS can also easily be interfaced, but this was not done in the timeframe of the thesis. An IMU can much easier generate reasonable values to test the filter, compared to a GPS where one has to be outdoors and move over a distance to gather data.

The MTi communicates over a 115200 BAUD serial line with a serial to USB converter. There is an internal Kalman filter which will not be utilized in this thesis, except as a source of filtered data for some illustrational purposes.

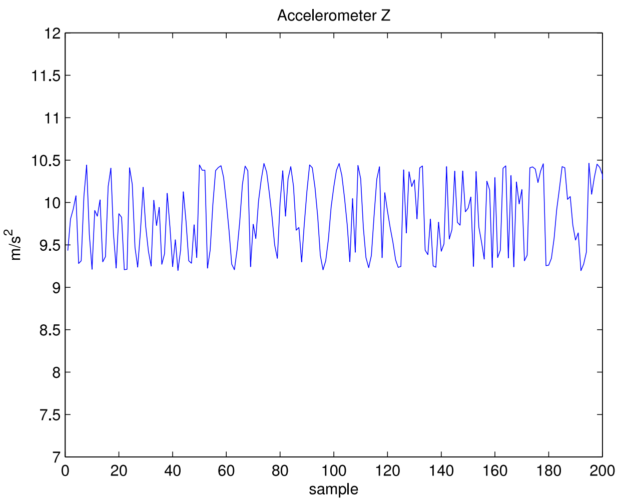


Figure 3.1: NORMAL

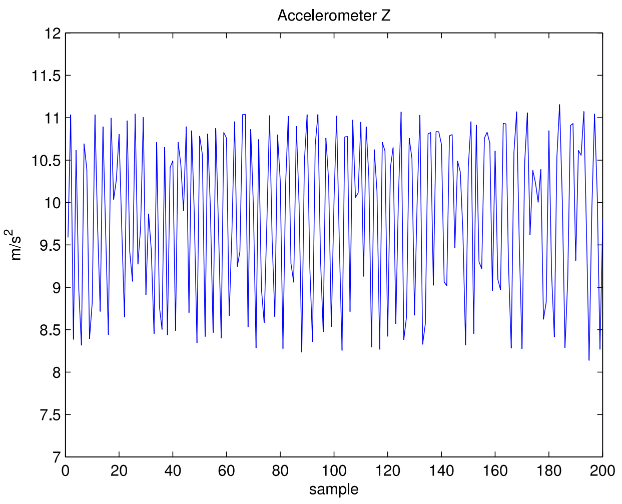


Figure 3.2: NOISE

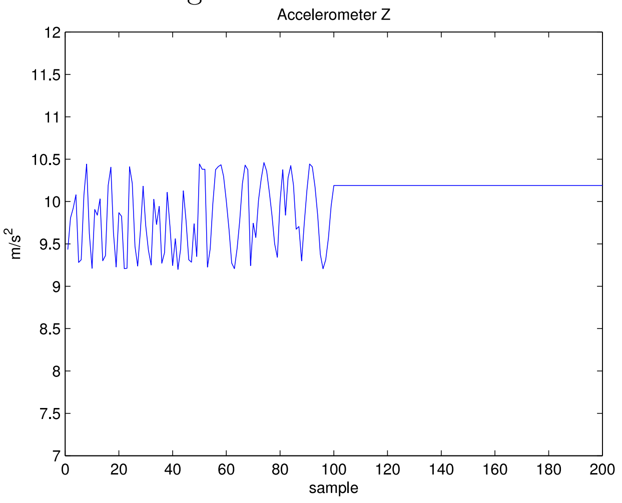


Figure 3.3: CONSTANT

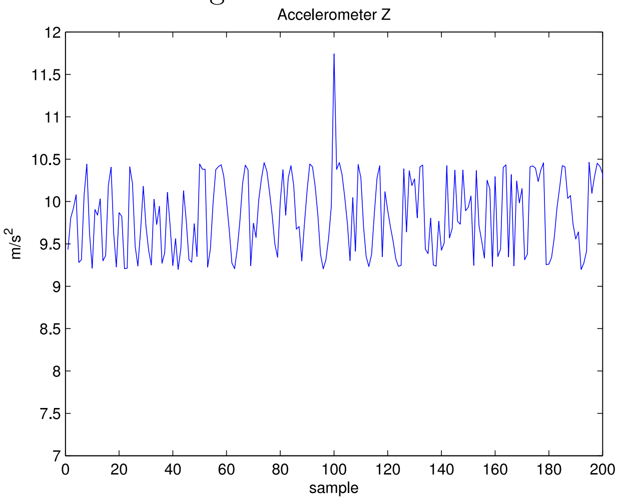


Figure 3.4: SHORT

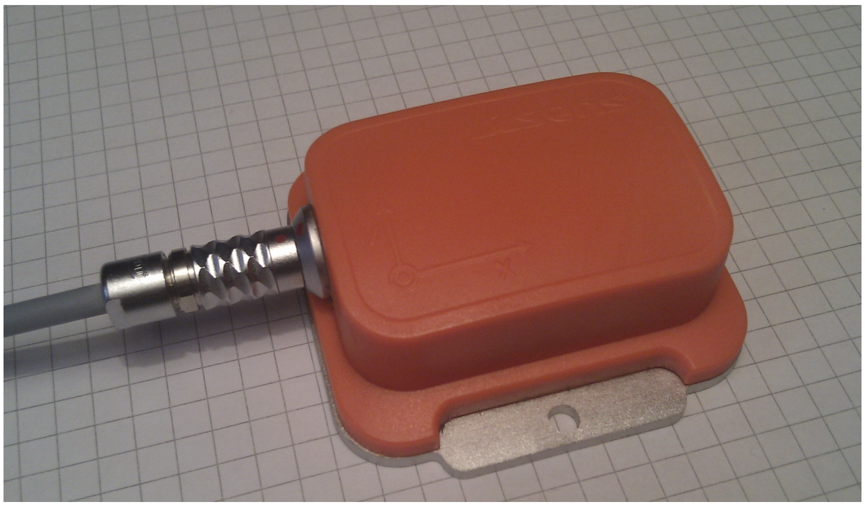


Figure 3.5: The Xsens MTi

The IMU was first tested with supplied files that display the IMU orientation in text according to user preferences to verify functionality. This functioned as intended, but some of these files are copyrighted in addition to providing an unnecessary amount of functionality, thus being unsuitable for further use.

Fortunately, [4, Ingebretsen] wrote a parser for the MTi fall 2010. These files combined with new code to access the serial port, provides all the IMU data in a manageable struct format. This code is now implemented in the software framework, so that the IMU is usable as a module both on a desktop system and on the BeagleBoard by changing the serial port between `/dev/ttyUSB0` and `/dev/ttyS2` respectively. Since the code has been interfaced to the framework, the IMU data is available in shared memory across the system in the following format:

```
struct struct_imu_xsens{
  long timestamp1_ms;
  long max_time_ms;
  unsigned long long missed_deadlines;
  long timestamp2_ms;
  char name_general[50];
  char name_specific[50];
  int mem_key;
  int pid;
  //--Module specific variables below--
  int file_in;
  int file_out;
  vec3_t acc;
  vec3_t gyr;
  vec3_t mag;
  quat_t q;
  vec3_t euler;
  float k_acc;
  float k_mag;
  float acc_cut;
  imu_quality_t quality;
};
```

Listing 3.1: IMU struct in shared memory

Notice how the first four fields are fixed in coherence with the framework rules as stated earlier in Section 2.4.2. The interesting fields are `acc, gyr, mag` which each contain unfiltered data in the form of three double values, one for each axis. `q, euler` contains Kalman filtered data in the form of a quaternion and Euler angles. The module specific fields are defined by [4, Ingebretsen], except for `quality` which is the topic of this chapter. The filter implementation will for test purposes use data from this struct.

3.4.2 MATLAB Graphics

In the package that is supplied with the MTi, there exist files for direct graphic presentation and MATLAB interface. Unfortunately, these files are only usable on a Windows platform, and the current BeagleBoard OS as well as the desktop OS are Linux based. To enable easier debugging, test the filter, and aid further development, having a graphical representation of the IMU orientation is essential.

By using and modifying the shared memory MATLAB interface presented in [1, Skøien, Vermeer pp.39 & Appendix H] together with the above mentioned files, it is possible to interface the IMU directly into Simulink. Since this interface is part of a previous report it is not commented here, but for easy reference, a slightly modified copy of the mentioned appendix is placed in Appendix D. The S-function was altered to read out the shared memory posted by the IMU module. This effectively makes Simulink a subscriber to the IMU in the software framework. The Simulink graphics are only used to read the shared memory contents on the desktop station, it is not under control of the master, nor does it post anything to shared memory. The code is available in the digital appendix A, and a guide to get the graphics running can be found in Appendix E. Consider Figure 3.6. The IMU appears as an ordinary block with exits for each of the sensors along all the rotational axis, as well as quaternion and Euler angles.

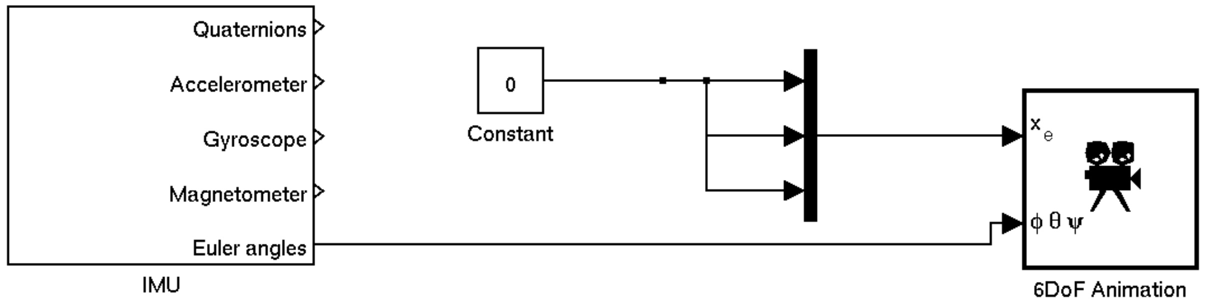


Figure 3.6: Simulink block diagram

The Euler angles are interfaced to a 6 DoF standard aerospace block, and the x, y, z craft positions are set to zero. As the simulation runs, it produces a smooth moving image of a missile that closely follows the physical movement of the IMU. See Image 3.7. This test is documented on video and is available in the digital appendix A. For a more realistic graphical representation, Dønnestad[3] produced an accurate model of the Recce D6 aircraft. The model was interfaced with Euler angles and a Simulink call to the "plot3D" function in the "reccePlot.m" file. This produced Image 3.8. Unfortunately, the draw function in the latter 3D plot demands more processing power than available, and is thus not good enough at displaying the IMU position in real-time. The 6DoF Animation block will be used to display live data from this point on.

3.5 Solution

3.5.1 General Structure

It was originally planned to have the signal quality detection as a standalone module in the framework. As implementation began, it became obvious that this would not be an optimal solution, since there is no guarantee that modules execute in lockstep (the detection filter could miss a sample). The filter block must run at the same frequency as the data provider, so that subscribing modules can read updated sensor quality data for each sample.

The solution is a general filter block that is an optional add-on to any framework module. It is written in C++ as a class, instantiated once for each measurement and called upon for each sensor cycle. As was pointed out in Section 2.3, making sure developers can write in their preferred language is an important feature. It must be emphasized that in contrast to the previous framework where developers were forced to write the entire module in C++,

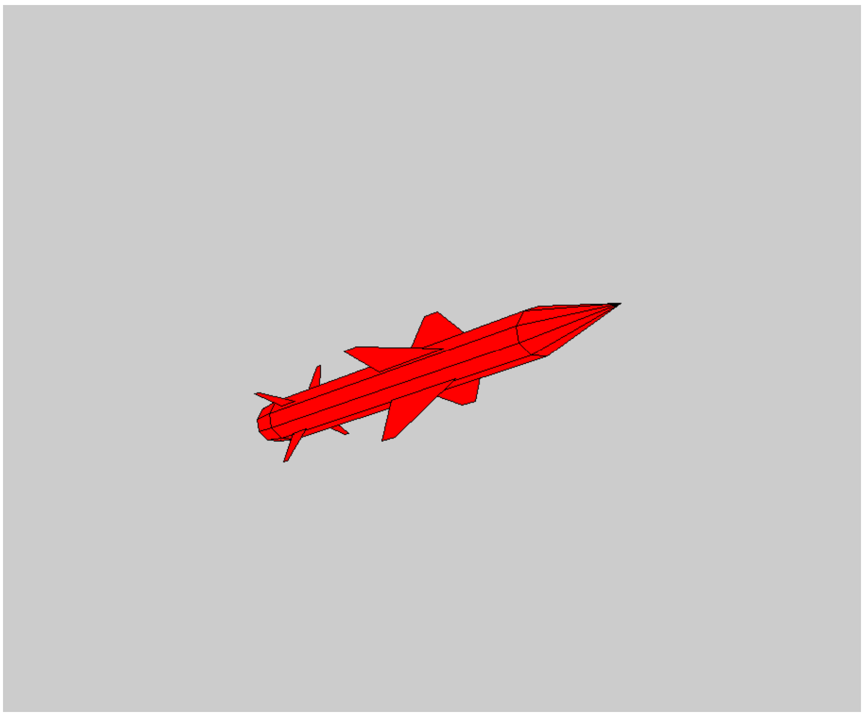


Figure 3.7: 6 DoF animation

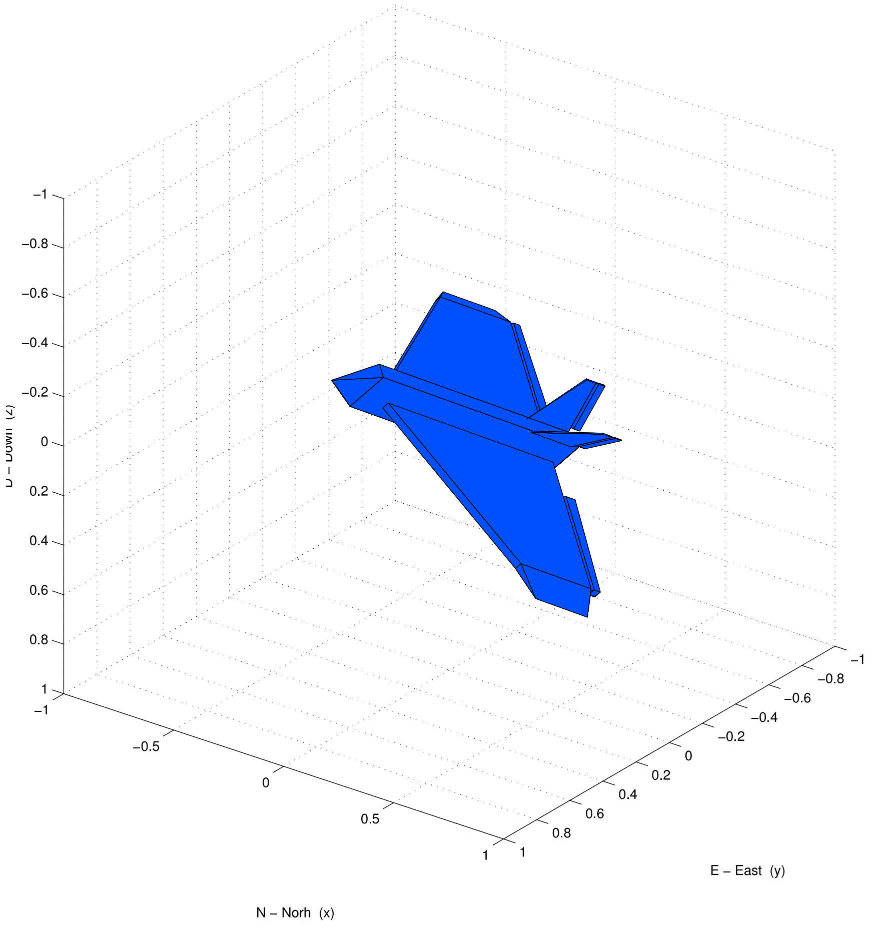


Figure 3.8: Recce D6 model by Dønnestad [3]

the added functionality of C++ in this case is contained within the functions and is not of developers concern. The one thing that must be done is create an instance of a class and call the filter function in the module's "while(1)-loop“.

All data from the sensor quality will be available in a struct that is posted to shared memory along with the sensor data and module information. For each output signal of a sensor, a struct is available for subscribing modules to read:

```
typedef struct sensor_quality{
    double variance;
    double average;
    bool flag_noise;
    bool flag_constant;
    bool flag_short;
    bool flag_high;
    bool flag_low;
}sens_qual_t;
```

To make certain that there is a logical way to read the sensor quality, it is stored alongside the sensor data in the globally available shared memory (Listing 3.1), and is accessed in the same manner as the sensor readings. To read data from the IMU z-axis accelerometer in a subscribing module:

```
imu_ptr->acc.v3
```

To read the variance of the same sensor:

```
imu_ptr->quality.acc.v3.variance
```

One could also store the quality struct together with the specific measurement, but with the current method, few alterations are required in the sensor module, and subscribers can easier filter out these quality data and choose whether or not to store them locally and use them.

3.5.2 Setting Filter Parameters

For each signal there are six parameters that must be set:

```
unsigned int samples;
double noise_variance_limit;
double constant_variance_limit;
double short_fault_limit;
double high_limit;
double low_limit;
```

Listing 3.2: Filter parameters

Where `samples` is the window size denoted N, `constant_variance_limit` is the lower variance limit before the sensor is marked as CONSTANT, `short_fault_limit` the maximum difference between two successive samples before the SHORT flag is raised, `variance_limit` is the NOISE flag limit, `high_limit` is the maximum value before the HIGH flag is raised and `low_limit` the opposite for the LOW flag.

These variables are read from a text file when an instance of the filter class is created:

```
Signal_quality filter_value_accv3("acc.v3",&struct_ptr_local->↩
    quality.acc.v3);
```

Listing 3.3: Creating a filter instance

Listing 3.4 depicts a typical text file containing filter settings for nine different measurements, such as made available through the Xsens's `acc,` `gyr` and `mag` readings.

```
This is the IMU filter parameters setup file.
int samples, double noise_variance_limit, double ↩
    constant_variance_limit, double short_fault_limit, double ↩
    high_limit, double low_limit.
--------------------------------
acc.v1 10 8 0.01 4 20 -20
acc.v2 10 8 0.01 4 20 -20
acc.v3 10 8 0.01 4 20 -20
gyr.v1 10 8 0.01 4 20 -20
gyr.v2 10 8 0.01 4 20 -20
gyr.v3 10 8 0.01 4 20 -20
mag.v1 10 8 0.01 4 20 -20
mag.v2 10 8 0.01 4 20 -20
mag.v3 10 8 0.01 4 20 -20
```

Listing 3.4: Example file format. (filter_settings.txt)

The first parameter to the constructor in Listing 3.2 is the keyword which is searched for in the text file, in this case `"acc.v3"`. The six numbers following the keyword in the text file is parsed and set as the filter parameters. By searching in this manner, the configuration file can contain other information, and it is easy for programs to write a configuration file, since the structure can be random as long as there is a keyword match followed by the filter parameters. It is also possible to update the parameters during runtime, by changing the contents of the text file and then stop (SIGTERM) or send SIGUSR1 to the module. One should use SIGTERM with great caution as this stops module execution, and is dependent on the watchdog timing out and rebooting the module. This not only takes time, but leads subscribers with old data for a prolonged period as well as severely impacting communication with hardware. Developers must evaluate this according to their module design. SIGUSR1 is a lighter way of updating the values. The SIGUSR1 signal is caught, a flag is set, the text file is reopened, and new filter parameters stored. Decreasing `samples` is safe, since the most recent samples are kept. Increasing `samples` will cause the most recent flag status to hold until N `samples` are once again stored in memory. The module could also read filter parameters from shared memory, but this requires some code modification.

Operators can now analyze data in real-time, adjust filter parameters accordingly and observe the results if two-way communication is available.

3.5.3 Excessive Long Term Noise (NOISE)

Noise is present in all types of measurements, and this is especially the case for instrumentation mounted in moving vehicles. Systems utilizing such data must be able to filter and cope with a certain amount of noise. Excessive noise (in comparison with what is common), could imply that a sensor outputs random values or some noise source has increased in amplitude. Examples can be an accelerometer too close to the motor, a sensor detaching from its damping bracket during flight, or a GPS loosing and regaining fix. Such faults can be detected by calculating the signal variance. Using variance to monitor signal quality was a solution proposed by the author early in the project, and is also utilized in [21, Sharma et al. pp.7], [20, Rogers pp.4267] and [23, Ramanathan et al. pp.8-9].

The variance across the user defined N last samples will be written to the quality struct, and can be used for e.g. weighing in observers. This can also be part of a log, and alert the operator of sensor faults.

The general formula for calculating variance in a selection is given in Eq. 3.1, and found in [24, Løvås pp.42].

$$s^2 = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2 \tag{3.1}$$

However, the implementation formula is structured in the following way:

$$s^2 = \frac{\sum_{i=1}^{N} x_i^2 - (\sum_{i=1}^{N} x_i)^2/N}{N-1} \tag{3.2}$$

The basic Eq. 3.1 assumes that the mean ($\bar{x}$) is known. In Eq. 3.2, this is calculated, and the two sums can run in the same for-loop, saving computation time. This is known as a *Naïve algorithm* since it is one of the simplest solutions. It is not the most accurate, but speed is a more important issue than accuracy in this implementation since the algorithm must run for each signal sample. The formula was for test purposes implemented in the following manner:

```
double Signal_quality::find_variance(void){
    unsigned int N = 0;
    double sum = 0, sum_sqr = 0, mean, variance;
    for(iter=prev_values.begin(); iter != prev_values.end(); ++iter)
    {
            sum = sum + *iter;
            N += 1;
            sum_sqr = sum_sqr + (*iter)*(*iter);
    }
    mean = sum/N;
    if(N == 1){
        variance = 0;
    }
    else{
        variance = (sum_sqr - sum*mean)/(N - 1);
    }
    return variance;
}
```

If the variance crosses the `noise_variance_limit` the NOISE flag is raised.

3.5.4 Dead Sensor & Frozen Signal (CONSTANT)

An apparent sensor failure is to provide no data at all. This can be due to the sensor itself failing, faulty wiring, driver problems or other issues. In some cases the module reads data from the sensor and uses the sensor itself to control the execution rate. The halt of sensor dataflow will in these cases also cause the module to halt, thus leaving the sensor quality detection useless. This scenario will be caught by the watchdog, mentioned in Section 2.4.7. Should the module execution not be affected by sensor timing, the quality detection comes into play. All the modules constructed at this stage (GPS & IMU) are built in such a way that the last transmitted sensor signal is stored for system-wide availability. Should the sensor fail, the subscribing modules will have access to the most current data. This should of course be used for calculations with caution, and the `quality` struct should always be read to verify sensor functionality.

The variance is also used to detect dead sensors. If the variance is below the `constant_variance_limit`, the flag_constant is raised. By setting this value to zero, the flag will be raised only when N successive equal samples are received. This low variance detection system is valuable in a range of sensors, e.g. a clogged pitot tube. Clogging renders the pitot useless, but even so, the value transmitted from the pressure sensor is noise prone and the samples will most likely not be exactly equal. Only checking if X samples in succession are equal to raise the constant flag is thus in many cases not good enough at detecting dead sensors.

3.5.5 Noise Spikes (SHORT)

Shorter physical jolts/noise or short communication errors leading to one or a few samples being numerically far from the mean value has to be caught separately from NOISE. This is due to the fact that variance is calculated from N samples to signal NOISE, and just one high sample will only give a small increase in variance. A flag needs to be raised immediately to alert subscribing modules of particularly high/low samples. To detect faults that fall under the SHORT definition, a method from [21, Sharma et al. pp. 7] is utilized, which in this paper yielded good results compared to other methods of SHORT fault detection. It is only a matter of comparing the two last samples, and determine if the difference is larger than a certain threshold.

3.5.6 High and Low Values (HIGH, LOW)

SHORT faults are calculated between two successive samples, but HIGH and LOW control the most recent sample to check if it is above the HIGH or below the LOW limit. Based on domain knowledge, developers can set limits which the sensor is unlikely or unable to pass in the specific application. As an example, a thermometer on an aerial vehicle should always transmit values within a physical feasible range.

3.6 Results

3.6.1 Test Conditions & Parameters

The following tests were carried out using data from the Xsens IMU connected to a desktop computer, whilst being manually agitated to simulate faults. Data is from the accelerometer z-axis, high flags are presented with the value "1", and marked with circles. The filter parameters are for illustrational purposes set to the following in this test:

```
N = 10 (window size)
noise_variance_limit = 8
constant_variance_limit = 0.01
short_fault_limit = 4
high_limit = 20
low_limit = -20
```

Figure 3.9 shows the S-function block which subscribes to the IMU. The first five outputs are for all sensors, while the remaining eight are specific for the accelerometer z-axis.

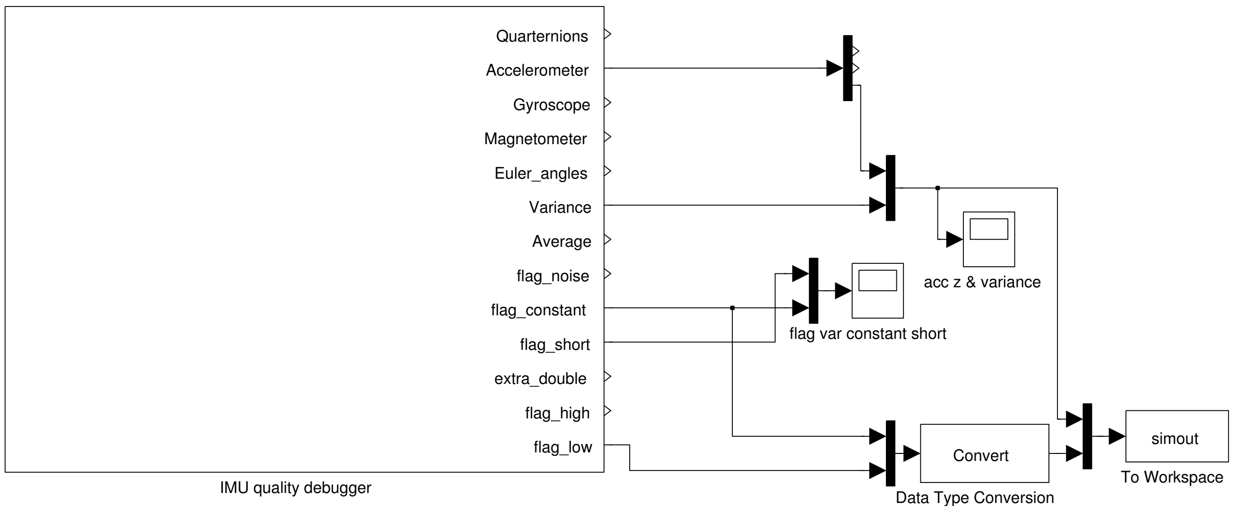


Figure 3.9: Simulink test setup

The test conditions and values are commented upon in 3.7.

3.6.2 Test Results

The NOISE flag and variance is presented in Figure 3.10. A slow roll is performed first, which yields a very low variance and no flags are set. The IMU is agitated harder along the z-axis, which gives a large increase of variance. When the variance crosses the `variance_limit` the NOISE flag is set.

The CONSTANT flag is tested in 3.11. A slow moving signal is present and then almost freezes at 760 samples, causing the CONSTANT flag to be raised when the variance falls below the `constant_variance_limit`. Note that the flag is raised even if are some modest variations in the signal.

In Figure 3.12 the IMU is stationary for the test, then briefly agitated in the z-direction. Three rapid jolts are given, where one is of magnitude large enough to trigger the SHORT flag. The `short_fault_limit` is set to 4 in this test, which equals: $4m/s^2 = 0.407g$ limit between two successive samples, which at 100 Hz sample rate equals a change of 0.407 g pr 10 ms.

The HIGH and LOW flag tests are depicted in Figure 3.13 and 3.14 respectively. The `high_limit` and `low_limit` are set to 20, -20. One can observe that the flags are raised immediately, at the same sample as the value crosses the limit.

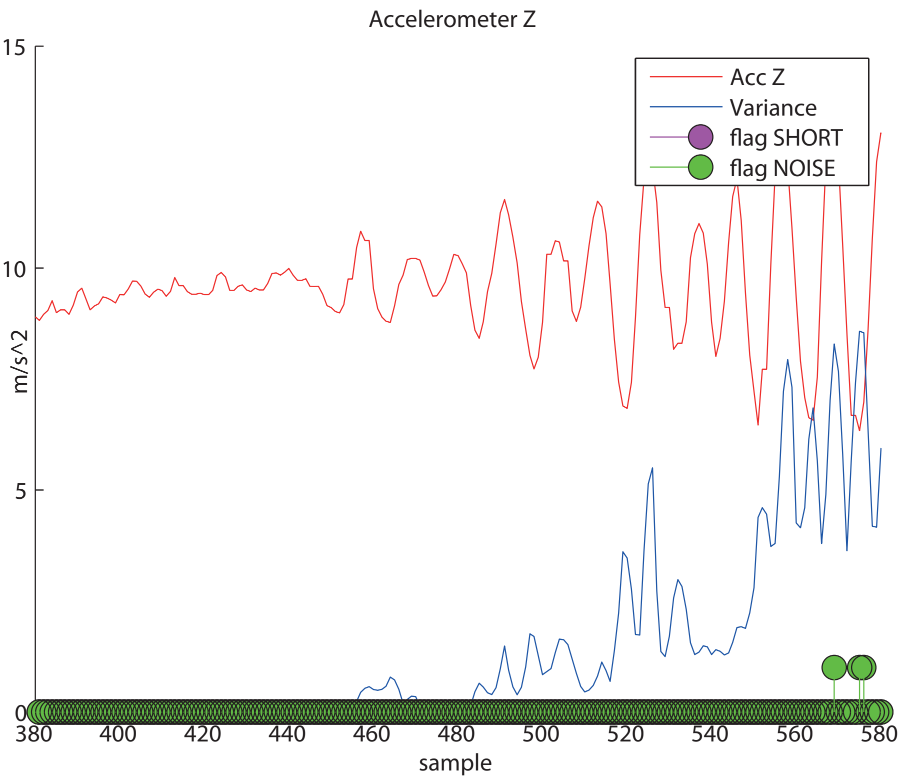


Figure 3.10: NOISE flag test

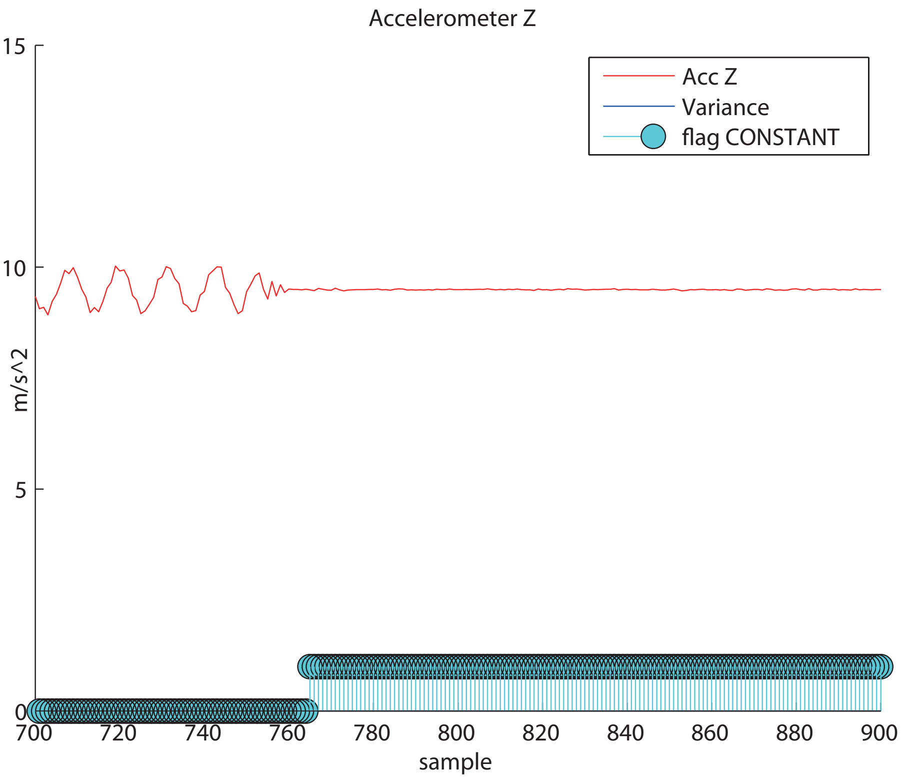


Figure 3.11: CONSTANT flag test

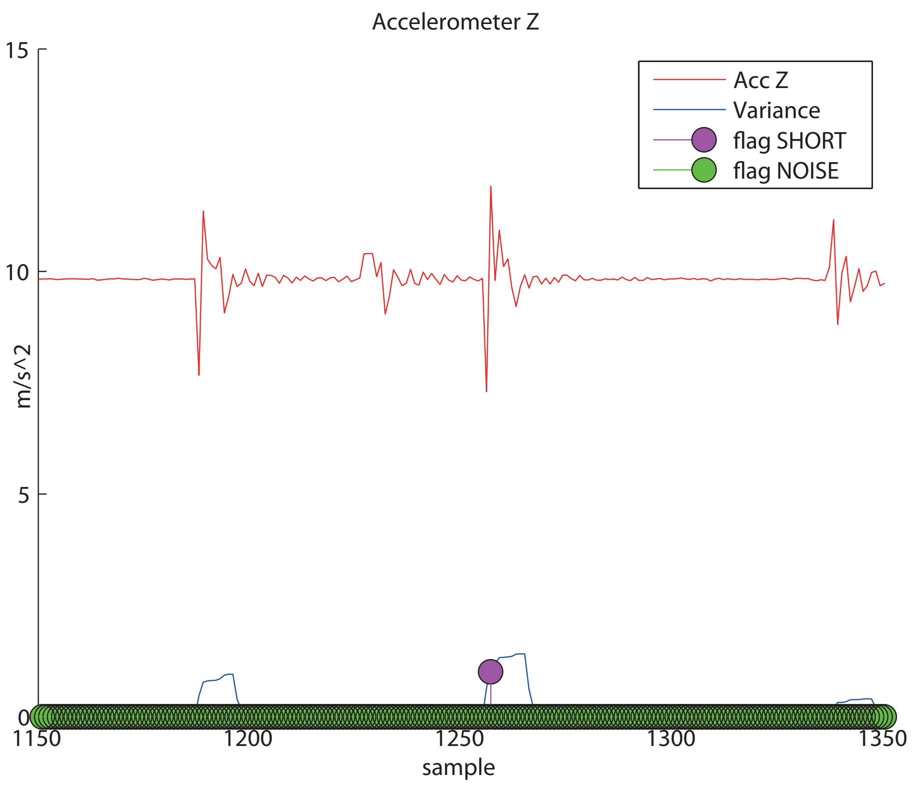


Figure 3.12: SHORT flag test

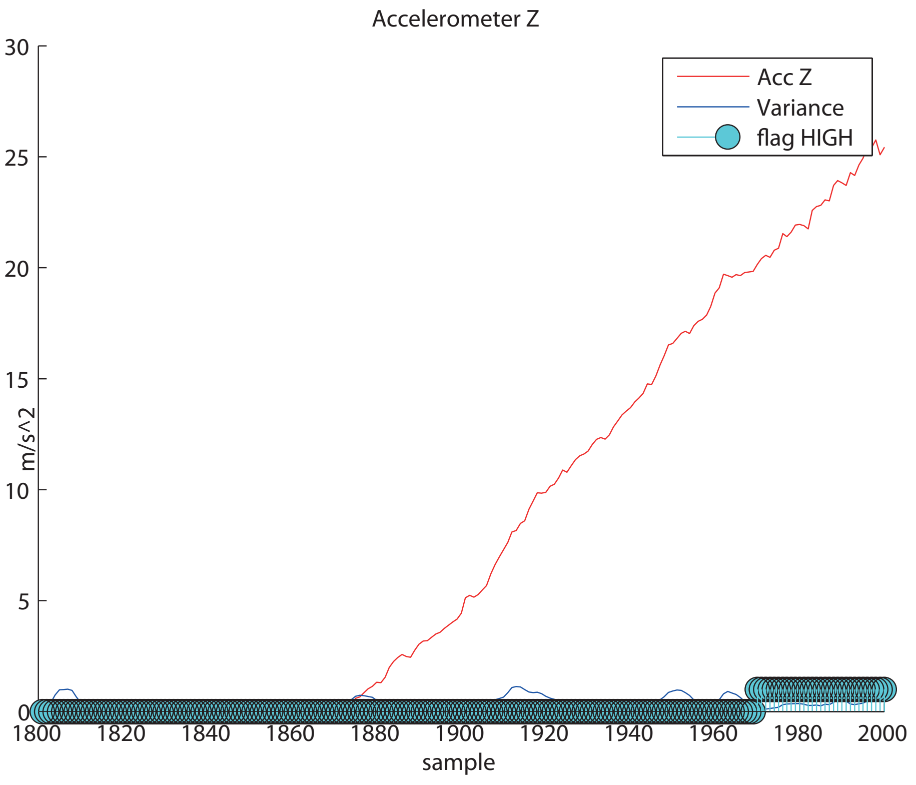


Figure 3.13: HIGH flag test

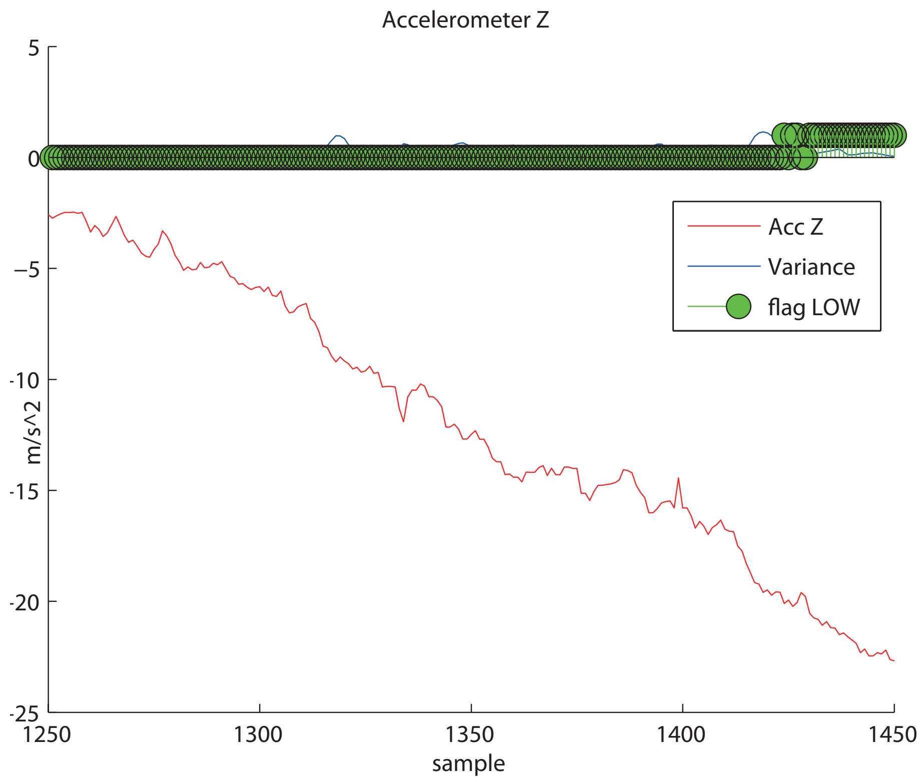


Figure 3.14: LOW flag test

3.7 Discussion

The above tests have been conducted in a controlled stationary environment, and the IMU agitations have been man-made and not produced by real-flight. Thus, the above is a test of the filter functionality and the numerical values are not representative of live data. To only test the filter, these induced values are quite sufficient.

The filter parameters are for this test set for illustrative purposes to verify filter functionality. When using the filter with live data, the parameters must be set according to sensor properties and ”domain knowledge”, which is emphasized throughout [21, Sharma et al.]. Testing the filter with real vehicle data, one can tune the parameters to give the optimum balance, detecting faults and minimizing false flags.

The filter proves to function very satisfactory in this controlled environment. Errors are detected correctly and flags raised according to their limits, without computationally heavy algorithms. Filter results are made available alongside the measured data in the software framework.

Chapter 4

BeagleBoard UAV Support Hardware

4.1 Background

For the framework to be capable of environment interaction, additional hardware is necessary to enable actuator control and sensor interfacing. As the system is targeted towards unmanned vehicles, servo actuators are commonly used for control (See Section 4.1.1). The servos will take place in the software framework as a module, and will ultimately subscribe to a control module's output to turn calculated values into physical actions. The hardware will also make possible for a wide range of sensors to interface the board, and they will as well end up as modules in the framework.

At the beginning of the semester project in 2010 [1, Skøien, Vermeer], a expansion board (*Trainer* from Tin Can Tools [25]) was ordered as an interface to the BeagleBoard to handle servo control. As the previous project neared completion, it became apparent that the Atmel ATmega328P processor on the Trainer Board was not dimensioned for the task. The servo resolution was low, and there was a desire to incorporate even more features. This chapter discusses the desired functionality of the expansion board, appropriate components, implementation and testing. This board is meant to be redesigned as the project progresses over time, as new demands arise. The reasons for utilizing an expansion board to handle certain operations are thoroughly discussed in [1, Skøien, Vermeer pp.45]. To summarize a few points:

- To accurately position servos, a high level of timing accuracy is needed. A dedicated processor is better suited for this task, as the BB is currently running a non-real-time OS.
- There are few vacant I/O pins on the BB.
- The BB I/O pins operate on 1.8 V, but most servos needs higher voltage for the control pulses.
- A cheaper expansion board will provide the BB with a protection barrier, since there are just a few connected bus lines with voltage level converters, and not a range of directly connected peripherals.
- The ability to manufacture the expansion board in house will support project longevity. New students can use the provided board design and code in this thesis and adapt it for new functionality.

- The expansion board method has proved to work well in many previous projects such as CyberSwan [26, Bjørntvedt E. pp.61-74], [27, Bristeau P.-J. pp.735] and in [28, Brekke S. E. pp. 43]. See Section 4.3.

4.1.1 Servos

A wide range of unmanned vehicles utilize servos as mechanical control organs. In a model size aircraft, servos are used for moving control surfaces, in cars they can be used for steering, helicopters for rotating the blades, just to mention a few examples. Providing the user with an easy interface for servo control is a desired feature of the platform.

Some information is provided in this section regarding the workings of ordinary RC servos and the pulse needed to control them.

Figure 4.1: Standard analog servo. Picture courtesy of sparkfun.com

An RC servo can turn a given number of degrees, usually somewhat more than 180° bank to bank. Due to the low gear rate, even small servos can pull and push with significant force. Normally a servo is connected by a +5V and GND wire for power, and a third low-current control pulse. The effect of this pulse can be seen in Figure 4.2.

The interval between the rising and falling edge gives the position set point to the servo. If there is 1.5 ms between the two flanks, this equals center and the servo will move to this position. A servo can be exited with pulse widths ranging from, 0.9 ms to 2.1 ms to use its entire region of movement. An internal potentiometer gives the actual position which the servo compares with the pulse and strives to keep the two equal. As long as there is no discrepancy between the set point and position, very little power is used. If force is applied to the arm, currents in the range of ~400 mA can be drawn as opposed to ~9 mA when idle.

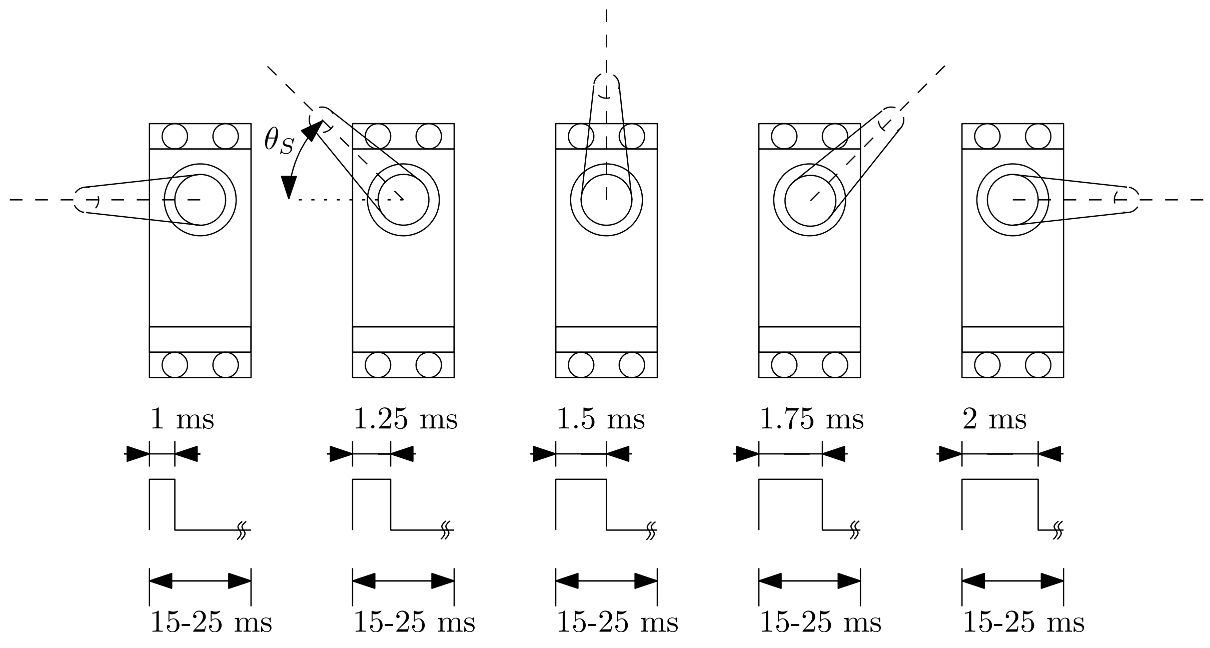


Figure 4.2: Servo control pulse

Both the range of movement, pulse width to position mapping, current draw, speed and other properties varies greatly across make and models.

4.2 Functionality

Before components can be chosen, one must define the desired functionality of the expansion board. Points here are based on the author's experiences in practice, earlier UAV projects such as those mentioned in Section (4.3), and points from Skavhaug and Fossen.

- Control of six servos
  The Recce D6 only utilize four (elevon1, elevon2, rudder and throttle), but in future models such functionality as flaps and landing gear may be present.

- Read servo positions in manual flight mode
  By registering what input is given to the craft as well as the response, one can improve and verify vehicle models.

- Auto/manual flight switch
  This enables the servo control to be switched from manual (operator) mode to auto. In many projects, this feature is realized in a microprocessor or programmable array logic. This has some advantages such as fewer components, lower cost and current consumption. On the other hand, these devices need power to pass the servo signal through. The proposed solution here is low-power mechanical relays to switch between flight modes. Should all power disappear on the 5V, 3.3V bus or HW errors occur, the relays will lose power and thus switch to manual mode. Servo power will be supplied directly from the engine speed control (ESC). One has to be vigilant with this design as well, since it is highly undesirable to exit auto-mode when the vehicle is out of range. It must also be possible to have some channels (servos) in manual mode and some in auto at the same time. This way the operator can control throttle, flaps and elevator while a heading (rudder, aileron) stabilizer is tested.

- Current and voltage measurement
  To inform the operator and control system of the remaining flight time, the power usage and remaining battery capacity must be known. This is seen as an important safety feature as well as giving knowledge of the impact of payload on flight-time. The user must inform the system of the type of battery pack used, and then via voltage and current measurement, power, energy used/remaining and minutes of flight time can be calculated.

- Provide stable power to the BeagleBoard
  With the benefit of increased functionality and computational power on the BB comes complexity and sensitivity. Should the power disappear or induce such a spike that the system reboots during long-range flight, the BB can use almost a minute to come back on-line. Although the Atmel processor can take certain actions to soften the landing without feedback, it is no match to the BB control system. The UAV is also a very noisy environment with magnetic fields, a range of electric frequencies and pulses, varying engine load and supply voltage. The control system supply must be noise-resilient, dependent and isolated as much as possible.

- Provide an interface to sensors such as IMU, ultrasonic altimeter and pitot
  Providing an easy interface to BB I/O such as SPI, I^{2}C and GPIO enables easy expansion to accommodate new units.

4.3 Previous Work

Many projects have before recognized the need to expand with additional hardware on the original system due to a variety of reasons. Issues that arise can be need for more I/O, more processing power, redundancy, real-time demands etc. As this project develops and hardware expansion is needed here as well, a study of earlier methods is executed to find the best solutions. Particularly good solutions that are suitable for this project are highlighted briefly in this section. Other points that influenced design choices are commented shortly. The authors comments on solutions are preceded by a hyphen (-).

The design that influenced the expansion board the most is the Trainer from TinCanTools. Website [25], schematics: [29]. The Trainer was purchased as part of the *General Platform for Unmanned Autonomous Systems*[1, Skøien, Vermeer]. It uses level converters to shift voltage levels between the BB and peripheral hardware. - This is also applicable on the expansion board. An EEPROM is used over I^{2}C to communicate with the BB to set the appropriate pinmux. - Such functionality is necessary in this design. The Trainer is equipped with an Atmel ATmega 328 processor. - A more powerful processor is desirable with more I/O.

The CyberSwan project has much in common with this thesis, as both utilize a Linux GPOS on a purchasable highly available platform [26, Bjørntvedt E. pp.12], and targeted towards unmanned aerial vehicles. [26, Bjørntvedt] expansion HW uses a very versatile DC-DC power-supply unit, the *Traco power TEB 15-2411WI*, which can operate in a wide voltage range and provide stabilized power with high efficiency. - A similar device is ideal for the expansion board. There is functionality that allows for servo signals from the receiver to be logged. - This is very valuable to examine inputs vs. vehicle movements in the lab to improve and verify vehicle models. The programmable array logic (PAL) circuit used to switch between auto/manual mode caused problems due to noise. - Power loss will also become an issue as the PAL circuit needs power to feed signals through. The Atmel ATMega128 and code used is programmed for a specific type of RC receiver. - The implementation must be able to accommodate any type of RC receiver.

[30, Veierland P. M. pp.14] does not utilize an expansion board, but two separate identical microcontrollers are used to handle critical and non-critical operations. - This dissertation provides detailed HW design schematics which are valuable to avoid basic design errors. The processors are part of the Atmel XMEGA series, and proved to work very well for this UAV purpose. - The XMEGA series are likely well known for many future developers in the unmanned vehicle laboratory due to the Atmel focus at NTNU, making it a very good candidate.

Two sources that have not contributed to the final design in a very considerable way are: [27, Bristeau pp.735] also develops a system with a central processing unit using mini-ITX, and a separate board for sensor interfacing. - This article does not go into depth of the design, but does provide some input on the overall design and peripheral interfacing.

[28, Brekke pp.27] Has the same layout with a central single board computer and I/O card. - This thesis does not provide specific design suggestions due to the PC/104 architecture, but stands as an example to the vast increase of available I/O due to an add-on board.

4.4 Components

The components chosen to fulfill the demands are listed in the following headings.

Focus is on low price and possibility to easily redesign and expand upon the expansion board. To ease development it must be possible to produce the board in-house. At time of writing, the tools available restricts the design to hand-solderable component packages and two-layer boards. As the project progresses, new users might create more advanced designs, and they must choose whether it is best to keep within the constraints of in-house production.

Common components such as resistors, connectors and LEDs are not a part of this section. These components can be found in the components list in appendix F. Board Design is commented upon after the components.

4.4.1 Main Processor

The lack of processing power in the ATmega238P is one of the reasons the expansion board had to be updated. Particularly the lack of accurate timers (16-bit) and corresponding interrupts complicated the code and made it highly integrated. The microcontroller demands are:

- Provide easy control and logging capability for at least 6 servos
- Have at least one UART for communication with the BeagleBoard
- Package must be solderable by hand
- Supply at least two analog to digital (AD) converters for measuring current draw and battery voltage

In addition, it is desirable to provide quite some overhead for future expansions. New users of the framework can reuse and modify existing code as well as alter the expansion board layout to accommodate new functionality. With the milling machine at the Cybernetics lab, producing a new PCB takes only a few hours from finished design to usable product.

The choice fell on the Atmel AVR XMEGA series, and more specifically, the ATxmega32A4. A TQFP44 package will be used as it is solderable by hand, and provides enough I/O to fulfill the demands. The 32 KB memory size was simply chosen due to the fact that this particular version was free of charge at the time of writing.

In addition to fulfilling the above mentioned criteria, Atmel controllers have been extensively used throughout the Cybernetics study, as well as the XMEGA series have proved its capability in several unmanned vehicles. (Local hawk Phoenix II [30, Veierland] and continued in Local Bug [31, Wenstad]).

The ATxmega32A4 has a significant amount of functionality, such as 5 16-bit timers, real-time clock, event system, and a range of communication links, ADC and DAC channels. Should even more functionality be desirable in the future, there are XMEGAs with larger package, more memory and peripherals available.

4.4.2 EEPROM

The Trainer board utilizes an EEPROM connected to the BB via I^2C interface. During boot, the BB reads the EEPROM to set the correct pinmux for the given expansion board. This design uses the same solution to set the pinmux, thus making sure the BB enables the correct pin functionality to utilize the expansion board.

The most viable solution is to use the same EEPROM as the Trainer, the Atmel AT24C01, since this is known to function properly. This particular item was not to be found, so a similar device, (M24C01-WMN6P) is used instead.

4.4.3 Voltage Regulators

In the specifications received from Odin Aero, the Recce D6 is equipped with an 8000 mAh/11.1 V Lithium Polymer battery (LiPo). A fully charged 3-cell LiPo reaches a maximum voltage of 12.6 V and a discharged (engine cut) voltage of $\sim$11 V. To maintain usability across a variety of vehicles, the power supply must be able to cope with even higher voltages. In helicopters, 6-cell LiPo packs are common, delivering a maximum voltage of 25.2 V.

On the output side, the BeagleBoard needs a stable, low-noise 5V DC power supply. Consult Table 4.1 for a rough current consumption estimate for the entire system.

As one can observe, the total system current draw is estimated to 734 mA. The servos which can be a major power consumer will be fed from the vehicles ESC. By having a separate supply system, the entire 5V (and thus 3.3V and 1.8V) can fail, disabling the BeagleBoard and expansion board all together, but still maintain manual control through the mechanical relays and RC receiver.

The *Traco DC/DC TEN8-2411WI* was chosen as the most appropriate candidate. Some of the main features of the TEN8-2411WI are:

- Voltage input range of 9-36 V

Item	Model	Typical current	Max current	Rated voltage
Main unit	BeagleBoard	270 mA	450 mA	5 V
EEPROM	AT24C01	5 mA	5 mA	2.7-5.5 V
Microcontroller	ATxmega32A4	13 mA	13 mA	1.6-3.6 V
IMU	ADIS16405BMLZ	70 mA	70 mA	4.75-5.25 V
GPS	EM-406A	44 mA	44 mA	4.5-6.5 V
Relay	G6K-2F-Y	21 mA*3	21 mA*3	5 V
Level converter	TXS0102	~0 mA	~0 mA	1.8 V & 3.3 V
Level converter	SN74AVC4T774	~0 mA	~0 mA	1.8 V & 3.3 V
Hall effect sensor	CSLS6B60	7 mA	9 mA	5 V
Camera	e.g.		50 mA	
RF module	e.g.		30 mA	
Servos	-	Ext. supply	Ext. supply	5 V
		Total:	734 mA	

Table 4.1: Expansion board current consumption

- 85% typical efficiency
- 5 V, 1500 mA max current
- -40°C to +85°C operating temperature
- 16 g weight
- Galvanic separation
- Shielded metal encapsulation
- Over-voltage and short circuit protection

There is also a need for 3.3 V and 1.8 V supply as well due to low voltage logic and BB interface. All the components using these voltages consume a very small amount of power, and thus more reasonably priced and accessible low dropout (LDO) linear regulators were chosen. LM1117MPX-3.3Ct-ND and LP2992IM5-1.8CT-ND will provide these voltages respectively.

4.4.4 Voltage and Current Measurement

A voltage sensor is accomplished by using the XMEGA ADC and a resistor divider network.

To log current consumption, a Hall effect method has been chosen. Instead of having a small resistor in series with the load, causing loss, the current is measured by letting the current pass through a magnetic field which exerts a force on the charge carriers. This generates a potential difference perpendicular to the current and magnetic field which can be measured. A wide variety of compact hall effect current sensors are available on the market, and to decide upon the model, a power consumption estimate was performed:

As stated in the Recce D6 specification, it is equipped with a 200 W brushless motor. 200 W/ ~12.0 V = 16.7 A. Even larger engines are quite common, with a current draw up to 50 A. The contribution from the control system is very limited: 0.734 A*5 V = 3.67 W 3.67 W/12.0 V/0.85 = 0.36 A current draw with 85% regulator efficiency. The hall-effect sensor must be able to measure in this range.

CSLS6B60 has been chosen as an appropriate sensor, due to its compact size, 60 A range and pre-amplified linear output.

4.4.5 Relays for Manual/Auto Switch

The relays used to handle the servo control switch must be monostable, be of low-weight, small size and have little coil current draw. Since mechanical relays have moving parts, they must also stand up to the G-forces and vehicle vibrations.

The G6K-2F-Y relay is ideal for this purpose.

- 1 g weight
- 200 G shock resistance
- -40°C to +70°C operating temperature
- Rcoil 237 Ω

Each relay has two switching contacts and can thus handle two servos.

4.4.6 Other Components

A range of level converters are used to increase and decrease the voltage, allowing for components and peripherals to communicate with the BeagleBoard. The UART, I^{2}C, SPI and some GPIO pins runs through level converters. Quite similar to the Trainer Board the Texas Instruments *txs0102* and *sn74avc4t774* are used for this purpose.

Such items as diodes, resistors, transistors, connectors etc. are chosen according to particular needs and are not commented upon in detail.

4.5 PCB Layout - Hardware Design

4.5.1 Demands

The PCB layout must fulfill the following demands:

- Low weight
- Possible to produce in house (max two layers, hand-solderable packages)
- Small size (keeping the form factor of the BB allows for easy mounting (3" X 3.1"))
- Provide an interface for IMU, GPIO, servos
- Provide the user with configurable LEDs for easy debugging

4.5.2 Solution

The board design was done in CadSoft EAGLE light edition, which is a familiar design tool to many fellow students and the author. Even more powerful design tools are available, but with price, easy accessibility and longevity in mind, it suits the project well. The light version can only use two layers, max 100 mm x 80 mm layout, supports only manual routing and only one sheet of schematics, which of none are limiting factors in this project.

The first board version (v1.0) proved to have too small via clearances and some of the mounts were a millimeter off. This, in addition to the router yielding bad results on a few wires, suggested a second attempt. Changes are summarized in Table 4.2.

Item	New Value
Moved BeagleBoard header to correct position	
Adjusted mounting holes	
Increased "via to wire" clearance	12 mil
Increased "wire to wire" clearance	8 mil

Table 4.2: Expansion board alterations v1.0 to v1.1

Version 1.1 turned out descent and it was intended to be the final prototype.

Soldering and tests went very well until the level converters were put in place. Some error caused the voltage on the 1.8 V rail to reach almost 2.3 V. At first, the 1.8 V regulator was changed, but with no results. Copper traces were severed until the txs0102 level converter emerged as the erroneous device. The component was switched without any success, switched again, this time rotated 180 degrees as it was unclear which pin was #1. This yielded no results, so a simple breakout housing only the txs0102 was made and tested. The connections were double checked by a peer, and even at this stage it drew almost 500 mA of the 3.3 V rail and acted irregularly. Upon closer inspection it proved that the wrong device was shipped. A very similar component package along with the correct information on the container caused the wrong device to be attached to the board.

At this stage the expansion board was in such a bad shape that the best solution was to make a completely new board. Version 1.1 still has a fully functioning XMEGA and relays, and will be used for testing and code debugging. Although this caused three to four days of extra work, it was also a good opportunity to do some other minor design alterations, summarized in Table 4.3. The final prototype version is known as v1.2. See Figure 4.3 and Figure 4.4 for layout 1.2.

Item	Value
Zener diodes switch to resistors due to high Zener current draw	
Increased some capacitor footprints	
Altered to "through-hole“ reset switch since this was available	
Increased the "wire to pad" clearance	10 mil
Increased the “via to wire“ clearance	15 mil

Table 4.3: Expansion board alterations v1.1 to v1.2

Due to the amount of components and small board size, the final layout has limited free space. The power supply plug and 5 V DC-DC regulator are placed close together and as far away as possible from sensitive components, such as the XMEGA. All the servos pass through centrally placed relays on the path through the board. Note that there is an additional control line into the board (seven instead of six) to enable auto/manual control. The servo control lines can be switched in pairs (1&2 3&4 5&6) between manual and auto mode. Should less than six channels be required for vehicle actuators, the remaining channels can be used to supply information to the XMEGA via PWM. There are three yellow LEDs along the right side to indicate individually that relays are in auto mode. Two

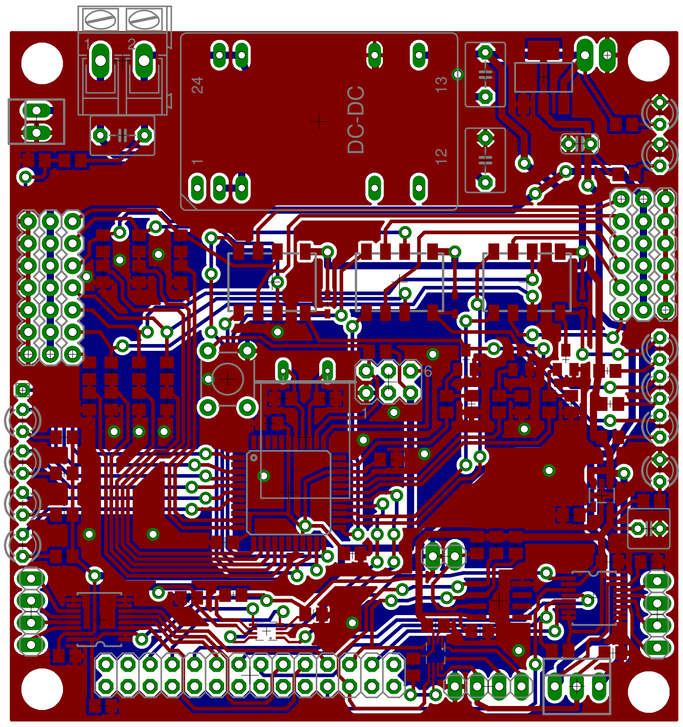


Figure 4.3: The 1.2 expansion board layout with traces visible. (Manually routed).

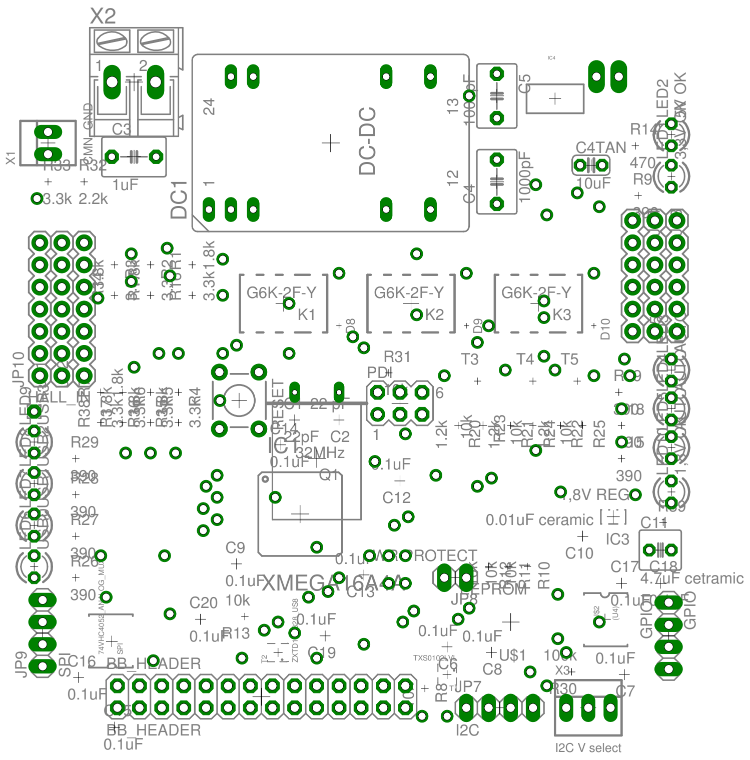


Figure 4.4: The 1.2 expansion board layout with names and values visible

green LEDs at the right top indicate that the 5 V and 3.3 V lines are powered. Four user-definable LEDs on the left side can be used for debugging and indication of control system status. The hall-effect sensor is supplied with 5 V, and the analog output connected to the HALL_IN pin. Battery voltage is read from a voltage divider into the XMEGA ADC. A Program and Debug Interface (PDI) and reset button can be found in the center of the board. Level converters for UART, SPI, I^2C and GPIO are placed close to the BeagleBoard connector. An EEPROM acts as a board identifier chip, transmitting information about the necessary pinmux setup to the BB upon boot.

As one can observe, there is no form for galvanic separation on this board. [26, Bjørntvedt pp.72] manufactured a similar expansion board, also without separation. Inquiries have been made regarding this issue, and as far as known, this should not cause any complications. No noise related errors have been experienced in any of the tests. Placing additional optocouplers on this two layer board to achieve galvanic separation would be a challenging task.

The board was made using a LKPF ProtoMat S62 mill, which takes appropriately 1.5 hours to mill a board of this size and complexity. It was checked for defects and soldered at the Department of Engineering Cybernetics (ITK) lab.

The interconnection principles can be viewed in Figure 4.5 and the finished board in Figure 4.6. Complete schematics can be found in appendix G. Eagle design files and Gerber files are available in the digital appendix A.

4.6 XMEGA Development

4.6.1 Peripheral Hardware and Programming Interface

Consider Figure 4.6 for this section, which depicts the board during the development process. All the items used to program and test the board are specified in greater detail in appendix H.
The board is supplied on the top left via a ∼13 V DC power supply. A standard 6-channel, 2.4 GHz, 3.3 V radio receiver is connected to the input and three servos on the output for testing.

The Rx and Tx pins are connected via the yellow and green wire to a STK500 development board for serial to RS-232 translation, used for debugging only.

In the center of the board a JTAG ICE MKII is connected to the program and debug interface, which in turn is connected to a windows PC via USB.

4.6.2 Code Development Environment

As previously stated, all the OS and HW related development on the framework is performed on an x86_64 Ubuntu GNU/Linux machine. "AVRDUDE" [32] is an open source tool available for this platform, suitable for programming Atmel devices. This is not an integrated development environment (IDE), and does thus not have the same user-friendly graphical interface and debugging tools. Atmel does provide developers with their own comprehensive IDE "Atmel AVR Studio" [33] which greatly simplifies the development and debugging process with features such as auto-complete, integrated compiler, correct syntax highlighting and built in debugger. Unfortunately, this software is available strictly for the Windows platform at time of writing. With the complexity and functionality demands of the expansion board, there will be much code on the chip. In the UAV lab with such a high mobility of the workforce, lowering the complexity level and providing new

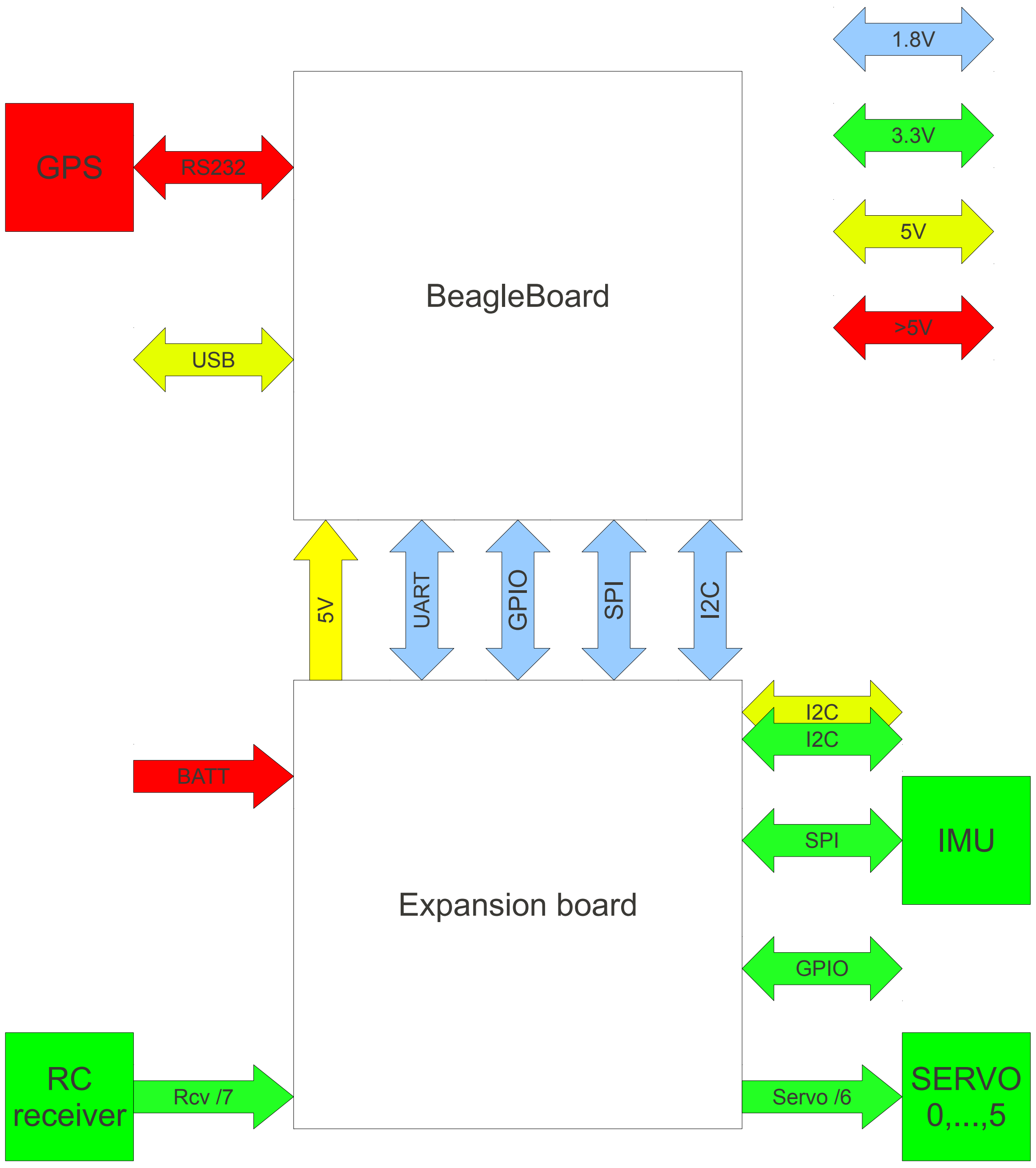


Figure 4.5: Peripheral connections

users with familiar tools is of high importance. Using some form of virtual machine on Ubuntu is also a possibility, but many people have negative experiences with this method. Therefore it was chosen to develop the code on a separate Windows 7 machine running Atmel AVR Studio 5.

4.6.3 XMEGA Code

This section gives a brief overview of the different files, and will not go into depth as the code is quite self explanatory and thoroughly commented. The amount of code makes it

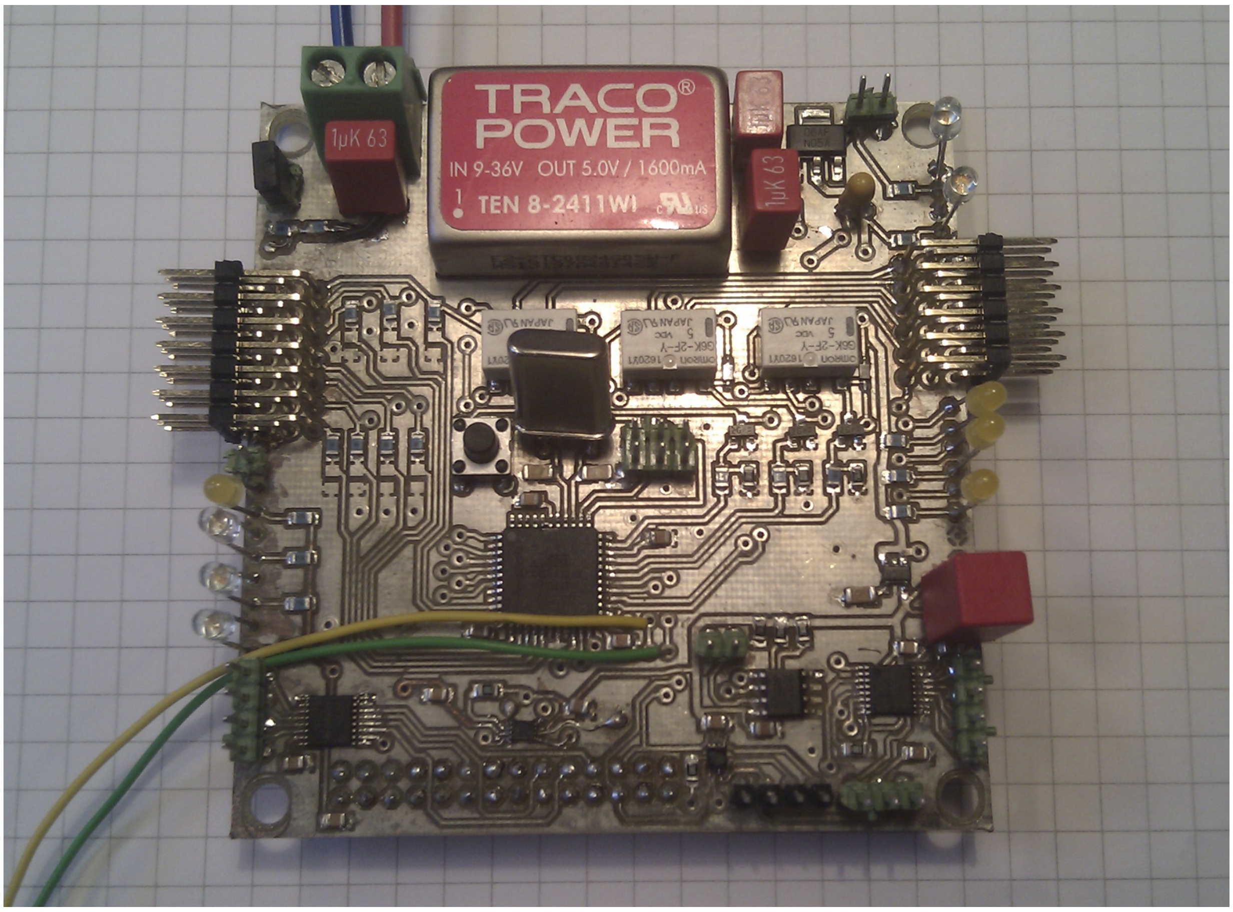


Figure 4.6: The finished expansion board

undesirable to include directly in the thesis. Snippets with belonging explanation can be found here, and the entire code can be viewed in the digital appendix A.
As mentioned, the code has been developed using AVRStudio 5 on a windows based machine. The code is designed very modular, with separate files for each function on the board. A given function is declared in the following manner:

```c
double adc_getvoltage(void);
```

Where the first word is the name of the general function, and the name of the .c file where the function is located. After the underscore is a more specific description of the function. By keeping to this strict formatting rule, it is easy to find the location of a function and understand what it does. Defines can be found in the appropriate .h file. Atmel provides a header file for each chip model that includes a wide range of definitions for the device. In this project, the file named "iox32a4.h“ is included via "avr/io.h" and is used to make the code as readable as possible.

Figure 4.7 illustrates the files used in the XMEGA project.

- `main.c`
  The main function is responsible for initializing the system and running the main program loop at a certain frequency. It begins by setting the appropriate system clock, and then calling upon all the setup functions.

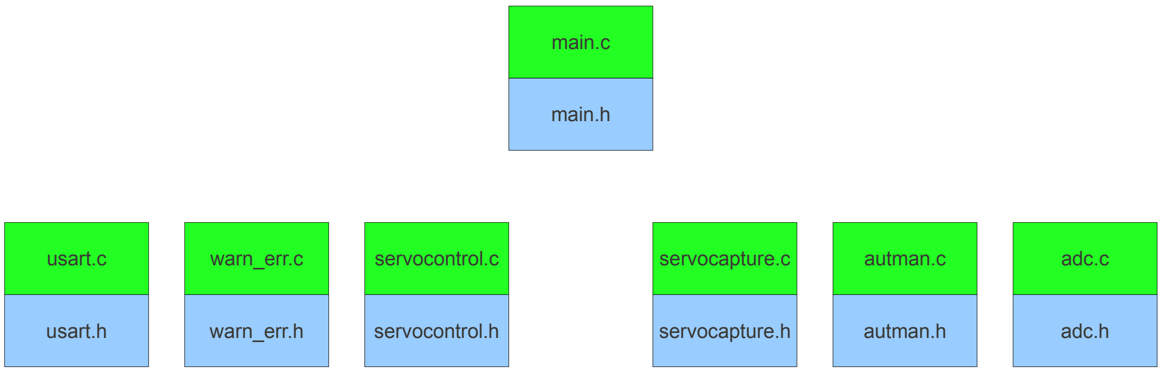


Figure 4.7: XMEGA code structure

```
main_ports_setup();
usart_setup();
warn_err_setup();
servocontrol_setup();
servocapture_setup();
autman_setup();
adc_setup();
main_startup_test();
```

Listing 4.1: Initialization in main()

Note how all functions cohere to the design rules. After the initialization, the master checks if a complete message is received over UART each iteration. If this is the case, that message is parsed and output to the servos if in auto mode. Main also checks if auto or manual mode is selected and sends desired data such as voltages, current and servo positions back over UART.

- `usart.c`
  As the name suggest, all the UART and printf() routines reside within this file. It configures the UART at the given baud rate, sets up interrupt driven receive and enables printf();. The latter is a heavy but powerful tool, enabling printf() with all its formatting capabilities over UART. Should the code become too large, it is recommended to remove printf() and replace it with a c-string print tool. In the given stdio.h library, the binary print function is not supported. Therefore, a custom binary print function was made, enabling easy viewing of uint8_t, uint16_t and uint32_t registers.

- `warn_err.c`
  The warning and error utility is a powerful debugging and status tool. As more files and functions are added, there is a need to standardize the handling of errors and warnings. A system was constructed where system warnings and errors are each stored in an integer, where each bit location equals a warning or error. Should a warning or error occur in a function, it reports this to the warn_err facility, and reports again when the issue is resolved. As an example, if the signal from the receiver disappears:

```
ISR(TCE0_OVF_vect){ // If the buffer owerflows = no signal in
  autman_pwm = 0; // Set to zero if timer overruns = auto mode.
  warn_err_seterr(NO_SIGNAL); // Report error
}
```

The `NO_SIGNAL` value is defined in warn_err.h, and is simply a given number of places to left shift a "1" bit. Warn_err stores this "1" in the given position and powers the red LED on the expansion board. By using the yellow and red LED for warnings and errors respectively, a user can easily see if the system functions as intended. The specific errors can be printed over UART by calling the `warn_err_print` function which uses the binary print utility to output the status.

- `servocontrol_setup.c`
  This file contains functions that parses the incoming UART message, and sets the timers accordingly to generate the PWM signals. There is no checksum in the message due to the short bus distance and the fact that at 100 Hz, errors will likely be corrected by the next message before the servos have been able to change their position significantly. The accuracy of the timer was a limiting factor on the previous solution, making the servo positioning inaccurate. With a range of 16-bit timers available, this is not an issue with the XMEGA. A 16 bit timer gives an overflow value of $C_{OVF} = 2^{16} = 65536$. Given that a servo control pulse period is 20 ms long, a clock divider must be chosen so there is no possibility of overflow, while maintaining the highest level of accuracy. $F_{CPU} = 32MHz$, the divider can be selected between $DIV = \{1, 2, 4, 8, 64, 256, 1024\}$ see [34, XMEGA manual pp.165], and $t_p = 20ms$.

  As can be seen in Eq. 4.1, a divider of 8 will cause counter overflow, thus a divider of 64 gives the best possible resolution. See Eq. 4.2.

$$t_p * F_{CPU}/DIV8 = 80.000 = C_{TOP} > C_{OVF} \tag{4.1}$$
$$t_p * F_{CPU}/DIV64 = 10.000 = C_{TOP} \not> C_{OVF} \tag{4.2}$$

  Given that the servo operates from max to max in the range of 1 to 2 ms, this gives a resolution of:

$$1ms * C_{TOP}/T_p = 500 \tag{4.3}$$

  Assuming that the servo operates from 0°to 180° this gives a rotational resolution of

$$180°/500 = 0.36° \tag{4.4}$$

  This is commented upon in Section 4.9.

- `servocapture_setup.c`
  Servo capture enables capture of the RC receivers PWM signals so that servo positions can be logged during manual flight. These channels can also be utilized for additional control of the XMEGA, given that those particular channels are not used for actuators. The positions are stored, and can be relayed back to the BB. Servo capture uses high and low flank interrupts from the RC receiver and compares it against a timer. Since these position readings are non-critical to system functionality, the lowest level interrupts are used. The XMEGA event system [34, XMEGA

manual pp.65] together with timers, has a special mode called “pulse width capture” which directly gives the pulse with in a simple and efficient manner. This is however not utilized here, since each pulse width capture require its own timer. The XMEGA 32A4 does not have sufficient timers available for this purpose, but the “auto manual switch” utilizes this feature as described later.

- `autman_setup.c`
  The auto manual switch uses a separate timer and event system to make use of the “pulse with capture” method to read the pulse from the receiver at servo_input 6. The event system does work without using the CPU, freeing valuable resources. Medium level interrupts are used since this is a critical function, which together with a separate timer enables very accurate readings of the pulse width. This means that this input signal can if desired be used for much more than just selecting between two modes, -auto or manual.

- `adc_setup.c`
  This file contains the setup method for enabling analog input reading on pin PORT A 5 and PORT A 7 for reading back the voltages from the hall-effect sensor and input voltage respectively.

4.7 Programming the Expansion Board EEPROM

The EEPROM is programmed in order for the BeagleBoard to set the appropriate pinmux. As an example, UART2 is disabled on the OMAP by default, but this is needed to communicate with the expansion board. At startup, U-BOOT reads from the EEPROM via I^{2}C to recognize the signature of an eventual expansion board and sets the pinmux accordingly. Since the expansion board uses the same pin configuration as the trainer, the EEPROM contents should be equal. “i2ctools” is an easy to use software tool available from the package system which reads and programs devices over I^{2}C. The contents of the trainer EEPROM were read, the trainer replaced with the expansion board and then programmed with the same contents. Upon reboot, the BB outputs: `Recognized Tincantools Trainer expansion board (rev 0 0)`. The EEPROM programming guide is placed in appendix I.

4.8 Testing

The expansion board interfaced to the BeagleBoard can be seen in Image 4.8.

The interface proved to work as intended. The BeagleBoard immediately found the EEPROM via I^{2}C, and the pinmux was set correctly. Messages over UART were both sent and received without problems. Auto/ manual mode functions properly, and the system enters auto mode if the receiver is turned off or is out of range. Servo control, servo capture and voltage readings all worked as intended. To test the servo resolution, a highly elongated servo arm was attached and the XMEGA solution compared with the RC receiver. There was no observable difference between the minimum deflection of 0.36° produced by this solution, and the smallest movement produced by the RC receiver. Since there was currently no need for SPI and GPIO, time was not spent investigating how to access these buses from the OS, and have yet to be tested.

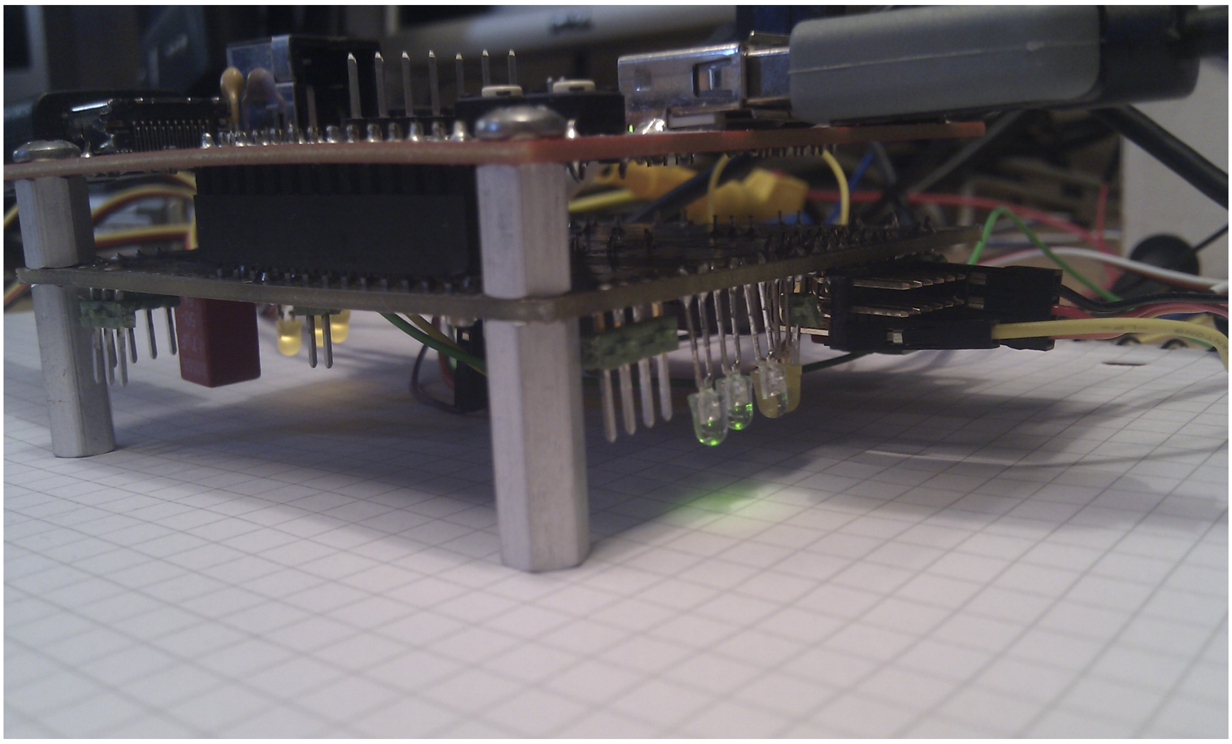


Figure 4.8: BeagleBoard and expansion board connected

4.9 Discussion

Due to the small size of some components, combined with only average soldering expertise, resulted in some of the solder joints being of mediocre quality. Under stationary testing this does not make any difference, but movement and vibrations during flight puts high demands on the joints. A final version might be produced to solve this and some other points mentioned in further work (Section 7.1.2). Since the expansion board is in its first versions, and is meant to be easily altered and added upon with project progression, it was highly desirable to produce it in-house to enable easy and fast re-design. A more dense and user-friendly board could be achieved by sending the design away for production with more layers and silk screen print. With the functionality now verified, there is less need for design modifications. New users may utilize and expand upon this design and choose whether to produce it in-house or send it away for production.

The given servo resolution of 0.36° is more than satisfactory based on [26, Bjørntvedt E. pp.54], and the fact that no difference in resolution was observed comparing the RC receiver and the expansion board. This is to some extent dependent on the servo model, and the specific servos used on the Recce D6 were not available for testing at the moment.

The warning error system creates a standardized way of providing the user with status feedback, both via use of LEDs and optional text over UART. This works well, and lets the user quickly identify issues in the system.

All the code on the XMEGA is structured and written in a very orderly fashion to allow for easy understanding and modification. The UART message is currently formatted for development and debugging, and is not very suitable for the finalized system, compared to a high-density but almost humanly unreadable binary format.

The expansion board itself solves the given demands well, is tested with good results, and is applicable in a wide range of vehicles. Recommended improvements to the board are gathered in Section 7.1.2.

Chapter 5

Full System Test

5.1 Aerial Vehicles

Although many topics of this thesis are applicable in general unmanned vehicles, it is worth to clarify some basic aspects of aerial vehicle orientation since this is commented upon later, and the system is tested and implementation is targeted towards aerial vehicle operation.

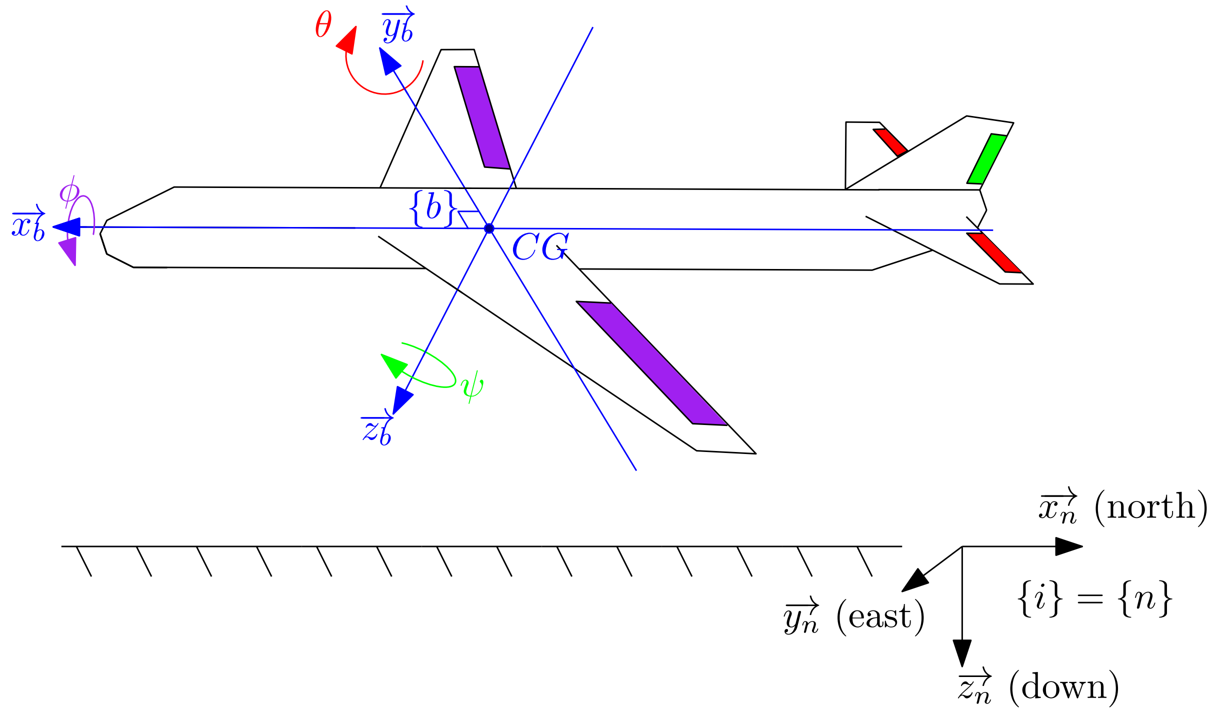


Figure 5.1: Aerial vehicle frames and rotation

Figure 5.1 depicts an aerial vehicle flying southbound and banking port. The vehicle has a fixed BODY frame ($\{b\}$) where the center is located at the center of gravity (CG), also referred to as the center of mass. The x-axis points in the longitudinal direction of the vehicle, y-axis to the right (pilot view) and the z-axis directly downwards perpendicular to the other two axes. This allows a rotation to be defined with regard to some inertial frame. Here the inertial frame ($\{i\}$) is chosen as the NED (north, east, down) ($\{n\}$) frame, which is the tangent plane on the earth's surface moving with the vehicle. $\overrightarrow{x_n}$ points to true north, $\overrightarrow{y_n}$ points east and $\overrightarrow{z_n}$ points down to the center of the earth normal to the surface. One should be aware that choosing NED as the inertial frame implies some simplifications such as ignoring the earth's rotation. This is beyond the scope of this thesis, as only vehicle rotations and some arbitrary position is of interest here. For more information see [35, Fossen T. I. pp.3-44].

Rotations in the figure are ϕ (roll), θ (pitch) and ψ (yaw); angles between the $\{n\}$ and the $\{b\}$ system. Each control surface is colored to represent the rotation it influences: Purple (ailerons, ϕ, roll), red (elevator, θ, pitch), green (rudder, ψ, yaw). This is a simplification for illustrative purposes, as for instance ailerons will cause roll, which in turn causes the airplane to lose some lift and influence pitch as well. Delta wings, such as the *Recce D6* combines the ailerons and elevator, and must thus mix the use of these combined rudders to achieve roll and pitch.

5.2 Feedback Loop Test

At this point, with all three main chapters of the thesis finished, the framework was tested to check functionality in a feedback loop system. The setup was made to simulate an aerial vehicle with stabilization on roll (ϕ) and pitch (θ), using separate servos for aileron and elevator control for illustrational purposes. The Xsens IMU was interfaced via UART to the BeagleBoard, and a simple P-regulator used in the loop. Figure 5.2 illustrates the general simplified system.

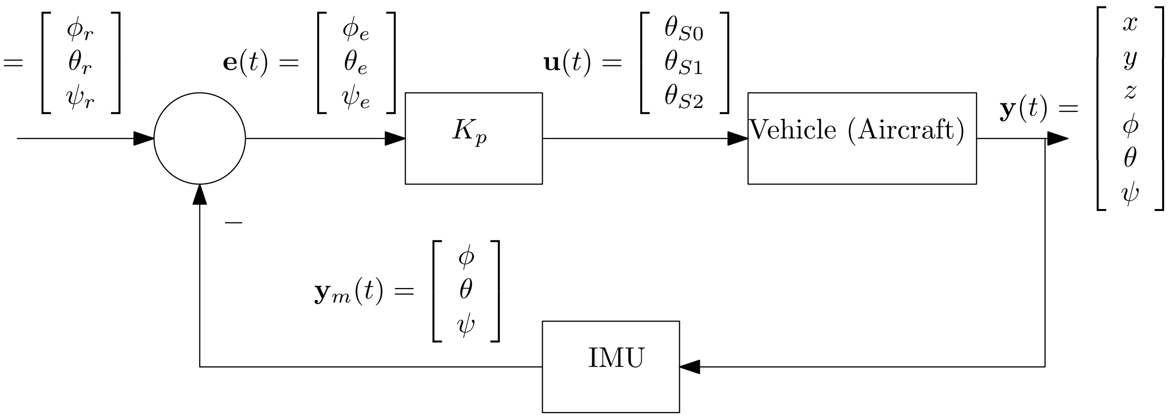

$\mathbf{r}(t) = \begin{bmatrix} \phi_r \\ \theta_r \\ \psi_r \end{bmatrix}$
$\mathbf{e}(t) = \begin{bmatrix} \phi_e \\ \theta_e \\ \psi_e \end{bmatrix}$
K_p
$\mathbf{u}(t) = \begin{bmatrix} \theta_{S0} \\ \theta_{S1} \\ \theta_{S2} \end{bmatrix}$
Vehicle (Aircraft)
$\mathbf{y}(t) = \begin{bmatrix} x \\ y \\ z \\ \phi \\ \theta \\ \psi \end{bmatrix}$
$-$
$\mathbf{y}_m(t) = \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix}$
IMU

Figure 5.2: Simplified feedback model

$\mathbf{r}(t)$ gives the reference which for this test is set to $\begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T$ to maintain level flight. $\mathbf{e}(t)$ is the error between the setpoint and measured vehicle attitude. $\mathbf{u}(t)$ is the servo input. The test does not include yaw stabilization, so there is no signal on θ_{S2}. $\mathbf{y}(t)$ describes the 6 directions of freedom (DoF), where x, y, z are some position from an arbitrary point on the ground which is irrelevant in a test regarding only attitude.

Figure 5.3 shows the test setup with control surface servos, the BeagleBoard with expansion board and the Xsens IMU. The IMU was interfaced as a module in the framework and the sensor detection filter used to catch signal anomalies. For test purposes the servo control module subscribed directly from the IMU module. In later stages of the project, the data flow will look similar to: IMU (and other sensors) $\rightarrow$ navigation (observer) $\rightarrow$ guidance $\rightarrow$ control $\rightarrow$ servos. The servo positions are sent in a preliminary test format over UART on

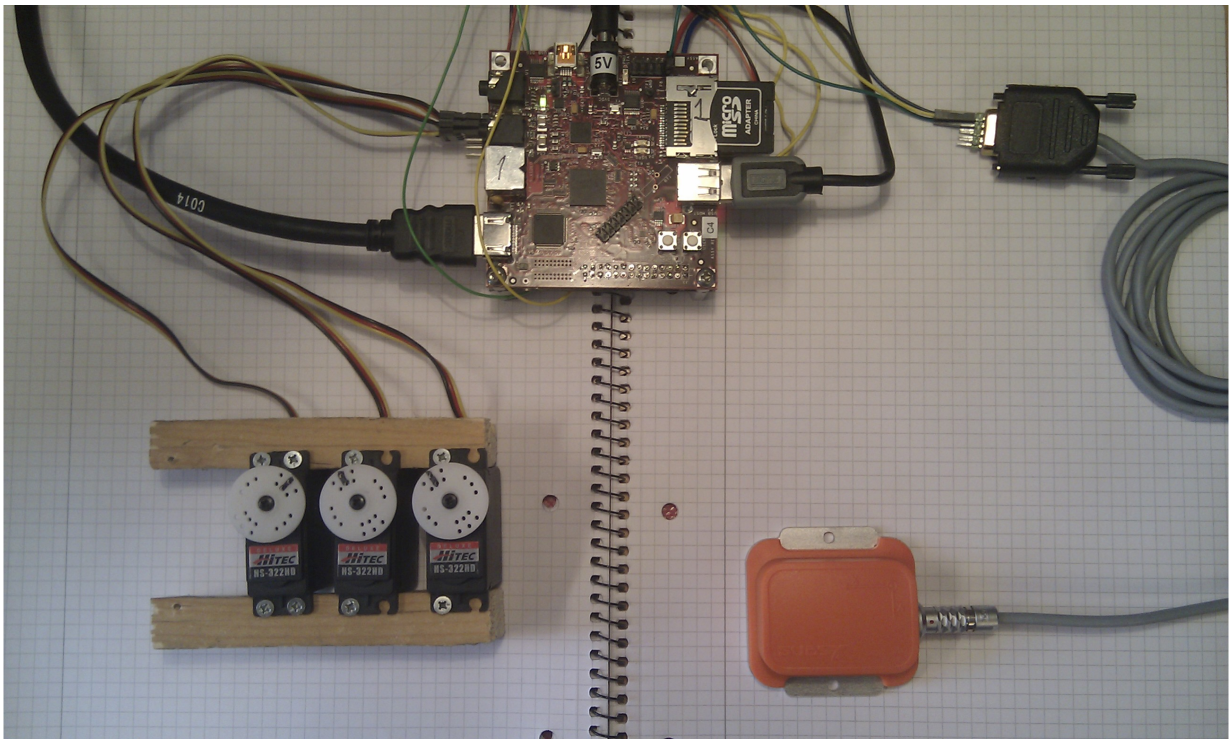


Figure 5.3: Setup for feedback test

the form: `500,1000,500,1000,500,1000,!`. Servo position string values are given in the range of 500 - 1000, are comma-separated and the string ends with an exclamation mark.

Currently the IMU and servo control runs at 100 Hz, and the servos at 50 Hz. The servos operate at highest possible frequency given the standard 20 ms pulse period. Files to recreate this test can be found in Appendix A. Run the framework as normal, according to Appendix B, and enable the Xsens and servo control module by entering the following in `system_master.h`:

```
string modules[] = {"./imu_xsens/imu_xsens","./servo/servo"};
```

The servos proved to respond rapidly to the IMU movements without any noise or lag.

Since the *Recce D6* is not available at time of writing, the test was confined to the lab, but none the less proving that all parts of this thesis works together in a feedback loop system. A video can also be viewed in appendix A.

5.3 Discussion

The feedback loop test shows that the results from [1, Skøien, Vermeer] together with the three main aspects of this thesis are able to work as a unit, providing a feedback controlled system. Measurements are received from the Xsens IMU, passed through the signal detection algorithm and the servo control module subscribes directly to the IMU data.

Since the *Recce D6* was not available at the time, no real flight test was conducted. This combined with the fact that the hardware is not exactly what will be appearing in the aircraft (servos, IMU); one can not draw any definitive conclusions on the capabilities of the system. On the other hand, due to the respectable specifications of the sensors and control system compared to other autonomous aerial control systems, such as [30, Veierland] and [36, ArduPilot Mega], combined with a successful preliminary test, the solution seems very promising.

Chapter 6

Overall Discussion and Evaluation

All discussions can be found locally at the end of their respective chapters. This chapter summarizes some points and performs a more general discussion of the solution, as well as a thesis reflection.

6.1 The Complete Solution

The described solutions in this thesis bring the autonomous platform up to a whole new level, and has successfully been tested for stabilized flight in the laboratory. It would have been an advantage to perform a real test flight as the most unforeseen issues have a tendency to appear when tested in the real world. To perform this test, a ground link would have been desirable to adjust parameters, some sort of casing for the hardware, and of course a suitable aerial vehicle to test with. The software and hardware are interfaced, and all measurements made available across the system, with optinal use of the signal detection filter. All parts of this thesis work well together, but new challenges will likely arrive when many modules created by different developers must work together. Although the software framework is made to assist in precisely such a setting, developers must still agree upon the data coming into and out of modules. - Even though the modularity and confinement is high, this does not help if module A uses degrees and B uses radians.

6.2 Thesis Reflection

When the original outline of the project was created early January, the planned content of this thesis was larger than the outcome turned out to be. Issues such as a battery model and interfacing the Analog Devices IMU were scheduled for completion. See Appendix J for Gannt charts. On the other hand, the thesis has successfully covered a lot of ground. As the project progressed, it became obvious that the software framework needed more attention, as more functionality was needed to become user-friendly and general enough. This was time well spent as the framework is now operational without any need of modification.

The next order of business was the construction of the filtering algorithm, which was a wish by Professor Thor I. Fossen. In retrospect it might have been better to leave this chapter for further work, as it is not crucial to test basic flight capabilities at this point. The saved time could have been spent further improving the framework, going more into depth, and implementing modules such as GPS and logging. Omitting this chapter would have allowed for studies on the topic of accessing SPI and GPIO from the BeagleBoard on the expansion board. This would also have reduced the number of chapters in the report, removing the least significant part. On the other hand, the work and results from the algorithm chapter with the interfacing of the Xsens IMU lead way for the final test, linking all parts of the thesis together. It also leaves the path clear for observers and control algorithms which will now be altered of faulty values. A graphical interface in MATLAB provides easier debugging and system visualization.

The expansion board is a crucial part of the total system, and went a few days beyond schedule due to the wrong device shipped from Digi-Key. The most central aspects of the board were completed on time and tested. As mentioned, a few days might have been spent investigating the GPIO and SPI access. This was not done due to time constraints and the fact that these peripherals are yet unused.

To summarize with the benefit of hindsight: The correct decision was made to phase out the battery and Analog Devices IMU interface. It would perhaps been better to only interface the Xsens IMU, but leave the filtering algorithm and graphics for later, enabling even more comprehensive work on the other two parts, while still performing the final test.

Chapter 7

Future Work

7.1 Existing Issues

7.1.1 Servo Module and Expansion Board Communication Format

The feedback loop test was performed using a preliminary communication format and a test servo module in the framework. New developers must decide on the communication format and finish the servo software module. By doing this, the servo module can subscribe to e.g. a control module and transmit positions to the expansion board, as well as making position readings, current and voltage measurements available to other modules in the framework.

7.1.2 Expansion board Alterations

As the expansion board was more extensively used, some additional points to improve appeared. Most design flaws have been corrected from version 1.0 to 1.2, but there are still some minor issues:

- One of the mounting holes connecting the expansion board to the BB is a millimeter off.
- A screw terminal should be added to provide easier access to the 5 V.
- The external 32 MHz crystal should be switched with a 16 MHz and run in 2X mode as stated in the XMEGA manual. ([34][pp.76].)
- PIN 40 (PORT A 0, AREF) should be connected to VCC.

7.2 Further Work

7.2.1 Test the Framework on Other OS

It is highly desirable to test the framework performance on a real-time Linux kernel; See [1, Skøien, Vermeer pp. 27] for tips on achieving real-time properties in Linux. Though only briefly studied, the Fast Context Switch Extension (FCSE) yielded improved context switch times on an OMAP1610 ARM v5 processor running Linux [37, Chanteperdrix, Cochran]. This might be applicable on the BeagleBoard OMAP3530 ARM v7 processor as well, according to the reference manual [38, ARM Architecture Reference Manual, Appendix E].

7.2.2 Additional Sensors

Many sensors might be interfaced to the framework, increasing the accuracy of the position and attitude estimation. Horizon heat, pitot, and ultrasonic height sensor are just a few examples. The most important sensor to interface is the Analog Devices IMU, which would significantly reduce the total system cost, as the Xsens MTi is a costly device.

7.2.3 Incorporate Observers

A range of observers have been constructed, both within the unmanned vehicle laboratory (See Section 1.3) and externally. These must be incorporated either via a MATLAB - Simulink module, or a module written in code which can interface to the framework.

7.2.4 Navigation and Control Module

No attention has yet been given to navigation and control modules. For any craft to become autonomous, these modules are essential. The GNC can be implemented as a single or separate modules, either via compatible code or MATLAB - Simulink. This can be done respectively by using the template files in the digital appendix A, or creating a Simulink solution similar to Figure 2.3 by using the guide in Appendix D.

7.2.5 Semaphore Capable of Addressing Multiple Readers-Writers Problem

At this point in time all the shared memory segments are protected by a binary semaphore. As stated in Section 2.5.2, if the number of reading processes are quite small (as they typically are in a UV), the reduced performance caused by an ordinary binary semaphore is negligible. Should the framework be used in a larger system where the number of subscribing (reading) processes is significant, a multiple reader-writer capable semaphore should be considered. This type of semaphore is also referred to as the “fourth semaphore”. By utilizing this lock, multiple readers can access the shared data simultaneously, while only one writer may write when no others have access to the region. This can yield higher system throughput as the readers no longer have to wait for each other.

Chapter 8

Conclusion

The unmanned aerial vehicle project at the Department of Engineering Cybernetics has taken a solid leap forward, with six fall projects and four master's theses completed on the subject within its first year.
Focus throughout has been modularity, low cost, and providing a software and hardware base that can grow with new system demands, knowledge and available technology.

A highly modular software framework has been developed, which accommodates hardware and software specific modules. The implementation is targeted towards unmanned vehicles, but the framework is usable in a range of applications. Since there is such a high level of abstraction, the solution is especially suited for ventures where there is a high mobility of the workforce, such as in student projects and theses. The framework has been run on the embedded BeagleBoard system, with modules for actuator control and attitude measurement. System overhead due to inter-module communication was found to be insignificant in a typical unmanned application. Context switches can become a limiting factor if the number of modules and their loop frequency is too high, due to the task switch overhead in the Linux kernel.

To detect errors in sensor equipment, an adaptable algorithm has been constructed that alerts subscribing modules of errors and anomalies in measurements. The filter parameters are adaptable during run-time, and such incidents as stationary signal, noise, spikes, too high or too low values are reported to other modules in the framework. This allows subscribing modules to take necessary precautions and react appropriately to different alerts.

Based upon earlier designs, new demands and ideas, an expansion board is fabricated for power filtering, actuator control and sensor connection, applicable in a wide range of vehicles. This has been interfaced to the BeagleBoard, providing protection, added I/O and a separate system for real-time critical actions. Positioning of servo actuators has been tested, and found to be accurate within $0.36°$, which is sufficient. The design is alterable as new demands appear.

A system feedback test has been performed, utilizing the software framework, signal filtering and the expansion board, positively verifying the functionality of all components.

The process of making a general platform for unmanned autonomous systems is indeed successful. All parts of this thesis have been linked, combined with earlier work, tested and run together as a single unit, in practice enabling autonomous operation for any vehicle given the appropriate control structure.

Bibliography

[1] K. R. Skøien and H. Vermeer, "General Platform for Unmanned Autonomous Systems." Department of Engineering Cybernetics, NTNU, Dec. 2010.

[2] K. Dønnestad, "Development of a flight simulator." Department of Engineering Cybernetics, NTNU, Dec. 2010.

[3] K. Dønnestad, "Simulation, Control and Visualization of UAS," Master's thesis, (in press), Department of Engineering Cybernetics, NTNU, June 2011.

[4] T. Ingebretsen, "Unlinear Attitude Observer and Hardware Interfacing for UAV Systems." Department of Engineering Cybernetics, NTNU, Dec. 2010.

[5] H. Nøkland, "Discrete Extended Kalman Filter for Inertial Navigation." Department of Engineering Cybernetics, NTNU, Dec. 2010.

[6] S. Hafslund, "Inertial Navigation System." Department of Engineering Cybernetics, NTNU, Dec. 2010.

[7] H. Nøkland, "Nonlinear Observer Design for GNSS and IMU Integration," Master's thesis, Department of Engineering Cybernetics, NTNU, June 2011.

[8] C. V. Stern, "Hardware in the Loop Framework for UAV Testing," Master's thesis, Department of Engineering Cybernetics, NTNU, June 2011.

[9] K. R. Skøien and A. Skavhaug, "A Highly Modular Software Framework Targeted Towards Embedded Applications Exemplified by UAV Usage." (unpublished), 2011.

[10] R. Garcia, L. Barnes, and K. Valavanis, "Design of a Hardware and Software Architecture for Unmanned Vehicles: A Modular Approach," *Applications of Intelligent Control to Engineering Systems*, pp. 91–115, 2009.

[11] C. Simmonds, Sept. 2009. `http://www.embedded-linux.co.uk/tutorial/periodic_threads` (last visited Jun. 19th, 2011).

[12] "Linux programmer's manual," Mar. 2009. `http://www.kernel.org/doc/man-pages/online/pages/man2/timerfd_create.2.html` (last visited Jun. 19th, 2011).

[13] R. Garcia, *Designing an autonomous helicopter testbed: from conception through implementation.* PhD thesis, University of South Florida, 2009.

[14] J. T. Holmgaard, C. S. Jensen, and S. Lænner, "Development and Navigation of an Autonomous UAV." Department of Control Engineering, Aalborg University, 2006.

[15] A. Burns and A. Wellings, *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX.* Addison Wesley, 2001.

[16] F. David, J. Carlyle, and R. Campbell, "Context switch overheads for Linux on ARM platforms," in *Proceedings of the 2007 workshop on Experimental computer science*, pp. 3–es, ACM, 2007.

[17] G. Heredia, A. Ollero, M. Bejar, and R. Mahtani, "Sensor and actuator fault detection in small autonomous helicopters," *Mechatronics*, vol. 18, no. 2, pp. 90–99, 2008.

[18] W. Flenniken, J. Wall, and D. Bevly, "Characterization of various IMU error sources and the effect on navigation performance," in *Proceedings of the Institute of Navigation GNSS conference*, 2005.

[19] B. Vik, "Integrated Satellite and Inertial Navigation Systems." Department of Engineering Cybernetics, NTNU, 2010.

[20] S. Rogers, "Sensor noise fault detection," in *American Control Conference, 2003. Proceedings of the 2003*, vol. 5, pp. 4267–4268 vol.5, june 2003.

[21] A. Sharma, L. Golubchik, and R. Govindan, "Sensor faults: Detection methods and prevalence in real-world datasets," *ACM Transactions on Sensor Networks (TOSN)*, vol. 6, no. 3, pp. 1–39, 2010.

[22] "Xsens website," (not dated). `http://www.xsens.com/en/general/mti` (last visited Jun. 19th, 2011).

[23] N. Ramanathan, L. Balzano, M. Burt, D. Estrin, T. Harmon, C. Harvey, J. Jay, E. Kohler, S. Rothenberg, and M. Srivastava, "Rapid deployment with confidence: Calibration and fault detection in environmental sensor networks," 2006.

[24] G. G. Løvås, *Statistikk*. Universitetsforlaget, 2. ed., 2004.

[25] "TinCanTools Trainer," 2011. `http://www.tincantools.com/product.php?productid=16149&cat=0&page=1&featured` (last visited Jun. 19th, 2011).

[26] E. Bjørntvedt, "Instrumentering av autonomt ubemannet fly: CyberSwan," Master's thesis, Institutt for Teknisk Kybernetikk, NTNU, June 2007.

[27] P. Bristeau, E. Dorveaux, D. Vissière, and N. Petit, "Hardware and software architecture for state estimation on an experimental low-cost small-scaled helicopter," *Control Engineering Practice*, vol. 18, no. 7, pp. 733–746, 2010.

[28] S. E. Brekke, "Autonomous Bicycle," Master's thesis, Department of Engineering Cybernetics, NTNU, Sept. 2010.

[29] TinCanTools, *TinCanTools Trainer Schematic*, trainer_rev-a1 ed., Apr. 2010.

[30] P. M. Veierland, "Local Hawk PhoenixII." Department of Computer Science, Aberystwyth University, Apr. 2010.

[31] P. Wenstad, "GPS Guided R/C Car," Master's thesis, Department of Engineering Cybernetics, NTNU, July 2010.

[32] "Avrdude," Jan. 2010. `http://www.nongnu.org/avrdude/` (last visited Jun. 19th, 2011).

[33] “AVRStudio 5,” 2011. `http://www.atmel.com/microsite/avr_studio_5/` (last visited Jun. 19th, 2011).

[34] Atmel Corporation, *XMEGA A MANUAL*, 8077h-avr-12/09 ed.

[35] T. Fossen, *Handbook of Marine Craft Hydrodynamics and Motion Control*. John Wiley & Sons Ltd., 2011.

[36] “Ardupilot mega,” 2011. `http://diydrones.com/profiles/blogs/ardupilot-mega-home-page` (last visited Jun. 19th, 2011).

[37] G. Chanteperdrix and R. Cochran, “The ARM Fast Context Switch Extension for Linux,” Real Time Linux Workshop, 2009.

[38] ARM, *ARM® Architecture Reference Manual ARM®v7-A and ARM®v7-R edition*, a ed., Apr. 2007.

[39] “Sourcery G++ Lite: ARM GNU/Linux: Sourcery G++ Lite 2009q3-67: Getting Started.” `http://www.codesourcery.com/`, 2009.

Appendix A

Contents of Digital Appendix

- Documentation
- Framework code
- Filtering code
- XMEGA code
- Eagle files
- Videos (MATLAB graphics test and feedback loop test)
- Report: K. R. Skøien and H. Vermeer, "General Platform for Unmanned Autonomous Systems." Department of Engineering Cybernetics, NTNU, Dec. 2010.

Appendix B

Testing the Software Framework (User Guide)

B.1 Introduction

To assist new developers with the software framework, this guide describes how to run a test utilizing the master module and two slaves on a Linux system. The files needed are supplied in the folder "framework" in the digital Appendix A. Here one can find the master module, dummy1 and dummy2 which are used for this test, the xsens_imu and the preliminary servo module.

To comment upon the two latter files, the xsens_imu is a finished file, reading data from the Xsens over a serial port. On a desktop Linux computer this is usually /dev/ttyUSB0 and on the BeagleBoard /dev/ttyS2 if it is connected to the BeagleBoard top serial header. The servo program simply subscribes to the IMU accelerometer, parses the message and sends a serial stream to the Atmel controller on the expansion board. This will thus only function properly on the BeagleBoard and is only there for illustrational purposes. If one wishes to create a new module, use a slave as a template, copy the code to a new file and customize from there.

B.2 Test File Functionality

The test setup is similar to the examples given in Section 2.4, where slave 2 provides data, and slave 1 subscribes to these data and prints to terminal. The files are configured as follows:

- system_master:
  Runs at 1 Hz, and prints information each cycle from both slaves in the format: name, memory key, PID, shared memory ID.
- dummy1:
  Runs at 1 Hz, reads a simple integer from dummy2 and prints the value to terminal.
- dummy2:
  Runs at 100 Hz, increments the integer by one for each cycle and posts its data to shared memory.

B.3 Compiling and Running

Move all the files in the `Framework` folder to your Linux computer. Make sure that `system_master.h` invokes the correct modules. The file should read:

```
string modules[] = {"./dummy1/dummy1","./dummy2/dummy2"};
//const string modules[] = {"./imu_xsens/imu_xsens"};
```

Each module has its own makefile associated with it which does all the compiling and linking of its respective module. The makefiles can be invoked individually by running `make` in the desired module folder. To ease the compilation process, a script residing in the `Framework` folder named `compile_all` will invoke all the makefiles.

```
user@user-desktop:~/framework$ ./compile_all.sh
```

The output should look similar to this:

```
======== Invoking all makefiles ========
====================
g++ -g -Wall -lrt -lpthread -o system_master system_master.cpp system_master.h↩
    timer.cpp timer.h
====================
make: Entering directory '/home/uav-2/masteroppgave/code/framework/dummy1'
make: Nothing to be done for 'all'.
make: Leaving directory '/home/uav-2/masteroppgave/code/framework/dummy1'
====================
make: Entering directory '/home/uav-2/masteroppgave/code/framework/dummy2'
make: Nothing to be done for 'all'.
make: Leaving directory '/home/uav-2/masteroppgave/code/framework/dummy2'
====================
make: Entering directory '/home/uav-2/masteroppgave/code/framework/imu_xsens'
make: Nothing to be done for 'all'.
make: Leaving directory '/home/uav-2/masteroppgave/code/framework/imu_xsens'
====================
```

For illustrational purposes, a change has been made in the system_master.cpp file so this is recompiled. The executables will now reside in the same manner as in Figure B.1.

Now, invoke the system_master and it will in turn execute the other modules

```
uav-2@uav-2-desktop:~/masteroppgave/code/framework$ ./system_master
```

The modules will begin their execution and print status messages. The master module should print messages at each cycle in the format described in appendix B.2:

```
MASTER: in while(1) ./dummy1/dummy1 1835890993 3910 2621473
MASTER: in while(1) ./dummy2/dummy2 1835890994 3912 2654242
```

The system runs in the manner described in Section 2.5, 2.6 and 2.7.

If it is desirable to compile for the BeagleBoard, consult appendix C and alter the makefiles accordingly. An example makefile can be viewed in the digital appendix A "makefile (crosscompile)".

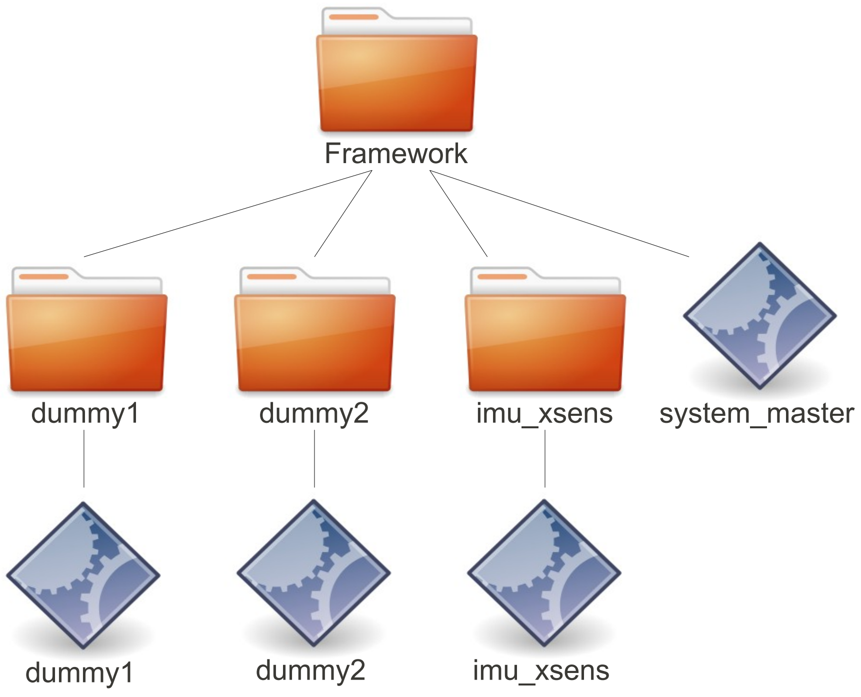


Figure B.1: Executables after compilation

Appendix C

Cross Compiling for the BeagleBoard

The following appendix is copied from [1, Skøien, Vermeer. Appendix G].
Follow this procedure to enable cross-compiling for the TI OMAP3530 ARM Cortex A8 (arm v7) processor.

1. Download the CodeSourcery command line cross-compiler:
   `http://www.codesourcery.com/sgpp/lite/arm/portal/package5385/public/arm-none-linux-gnueabi/arm-2009q3-67-arm-none-linux-gnueabi.bin`.
   Consult `www.codesourcery.com` for the newest lite version.
2. Make sure the .bin file is executable by using chmod.
3. Run the installer in the xterm shell.
4. After installation, add G++ to your respective path. Example:

   ```
   export PATH=$PATH:/home/uav-2/CodeSourcery/Sourcery_G++_Lite/bin/
   ```

   Remember that each time you close the shell, the path has to be added again.
5. Verify the install with `arm-none-linux-gnueabi-g++ -v` The last line, depending on the version, should read `gcc version 4.4.1 (Sourcery G++ Lite 2009q3-67)`.
6. At this stage, a cross compiled program will run on target with the exception of some flags which are not yet recognized. Preferably test the installation by making a simple "hello world!" program and run it on target.
   `$arm-none-linux-gnueabi-g++ -o hello_world hello_world.cpp`
7. To achieve full support for hardware floating point (-mfpu and -mfloat-abi flags), some additional runtime dependencies must be transferred to the BeagleBoard.

   See the manual [39, pp.14-18] for several ways of transferring and using the required files. As an example, we assume that it is desirable to:

   A. Install the dependencies in a separate folder. It is possible to move the runtime support files to the BeagleBoard base files, but this might cause the BB not to boot properly.

   B. Passing arguments at compile time to provide information where to find the runtime support files. If the files were installed in the root folder, this would not be necessary. The user can choose to provide the paths at runtime as-well, but this requires quite some text to run the executable.

8. Find the folder `../CodeSourcery/Sourcery_G++_Lite/arm-none-linux-gnueabi/libc/thumb2/` on the host computer. To get compiled code running, copy the subfolder `/lib` to the target, e.g. to the folder `/root/sysroot/thumb2/lib`. Note here that `/root/` here is the root user, not the file system root. The `.so` files in `../thumb2/usr/bin` & `../thumb2/sbin` should also be copied into their equal sub-folders under `/root/sysroot/thumb2/` on target. There should now be three sub-folders in the thumb2 folder. The mentioned manual provides more detailed information about which files one needs to move to target. At compile time, the user must provide information to the executable where to find the runtime support files. In our example this is done in the following manner: `arm-none-linux-gnueabi-g++ -mtune=cortex-a8 -mfpu=neon -mfloat-abi=softfp -Wl,-rpath=/root/sysroot/thumb2/lib: ↩ /root/sysroot/thumb2/usr /lib -Wl,--dynamic-linker=/root/sysroot↩ /thumb2/lib/ld-linux.so.3 -o master master.cpp`. Additional flags may be applicable.

9. Copy the executable to target.

10. Run your executable like normal with `./system_master`.

Appendix D

Interfacing Simulink with Shared Memory

The following appendix is a modified version of [1, Skøien, Vermeer. Appendix H].
This guide is written as an introduction to passing data between Simulink and legacy code.

1. You must have MATLAB, Simulink and Real-Time Workshop installed on the host computer. This method is verified with MATLAB version R2010a 64-bit (glnxa64) Ubuntu GNU/Linux, Simulink Version 7.5 R2010a and Real-Time Workshop version 7.5 running with ITK site license.

2. In Simulink, find the S-function builder block. This block enables hand-written C/C++ code to be implemented into Simulink.

   The S-function can also be written by hand, but it is recommended to at least use the builder once to see the C/C++ file structure. Since data will be passed to Simulink, treated by the diagram and then passed back to the process, HW interfacing blocks should ONLY have outputs OR inputs. This coincides with the way Simulink executes the blocks, one at the time from input to exit. Entering the builder block, choose an appropriate name for your S-function. Under "initialization", choose sample time = inherited, unless this block needs to run at a different frequency than the main Simulink loop. Select "data properties" delete all inputs (read from HW) and set up the desired number of outputs. The default data type is double with one-dimensional ports. Change this according to spec. Under "build info" unmark the checkbox "Generate wrapper TLC" and Press "build". The S-function will be generated in the working directory. There is no longer need for the S-function builder, delete this and add the block called "S-function". The latter block will act as a hardware abstraction layer (HAL), providing data to Simulink.

   In the S-function block, write the name you gave in the "S-function name" field. Press "edit" to make sure that the C/C++ file is found. Leave "S-function parameters" blank, and leave "S-function modules" = "".

   To verify the function of the code, it is recommended to put a *printf("running\ n");* in the MdlOutputs function in the code. Since it was chosen to not generate a wrapper, the line that calls "outputs_wrapper()" in "mdlOutputs()" must be commented out.

   Running the simulation demands some adjustment. Under Simulation $\mapsto$ configuration parameters choose solver. Set the desired stop time (e.g. 10.0 s), type: ”fixed step“, Solver: ”discrete“, ”fundamental sample time“ according to desired loop frequency e.g. 0.1 s.

Before running the simulation the C/C++ file must be compiled. This is done with ”mex <yourfilename> in the MATLAB command. Simulation$\mapsto$ start should now print “running” 10s/0.1s = 100 times.

Appendix E

Running the IMU with Simulink Graphics

This guide describes how to get the graphical interface from Section 3.4.2 to run with the supplied files.

1. Ensure that all the software mentioned in Appendix D is installed. This guide has been tested with the above versions, but should work on newer releases as well.
2. In the supplied folder `Framework` open the `system_master.h` file. This must be configured to read the IMU, and should thus look similar to:

```
.
.

//string modules[] = {"nice -n 5 ./dummy1/dummy1","./dummy2/↩
    dummy2"};
string modules[] = {"./imu_xsens/imu_xsens"};
.
.
```

3. Compile the modules using `./compile_all`.
4. Connect the Xsens MTi via USB.
5. Run the framework with `./system_master`. The framework should now be reading the IMU, and the master will give periodic execution messages.
6. Start MATLAB with root privileges.
7. Open the supplied folder `Matlab` and add it to path.
8. Run `mex imu_input_framework.c`. This should compile with no errors.
9. Open the Simulink model `imu_graphics.mdl`.
10. Make sure that the framework is running and start the simulation. The missile graphics should appear and follow the IMU movements.

Appendix F

Expansion Board Components

Item	Producer	Description	From	Orderno	No.
EEPROM	ST	M24C01-WMN6P	Elfa	73-970-11	1
Microprocessor	Atmel	ATxmega32A4	Omega V.		1
Switchdiode	NXP	0.25 A	Elfa	70-187-73	3
DC DC converter	Tracopower	TEN 8-2411 WI	Elfa	69-538-06	1
Power connector		MKDS 1.5/3-5.08	ITK		1
Relays	Omron	G6K-2F-Y	Elfa	37-022-48	3
Resistor		470Ω, SMD	ITK		1
Resistor		1.8 kΩ, SMD	ITK		7
Resistor		10 kΩ, SMD	ITK		9
Resistor		3.3 kΩ, SMD	ITK		1
Resistor		22 kΩ, SMD	ITK		1
Resistor		1.2 kΩ, SMD	ITK		3
Resistor		390Ω, SMD	ITK		7
Transistor		BC850C, SMD	ITK		3
LED		Yellow, 3 mm	ITK		4
LED		Green, 3 mm	ITK		3
LED		Red, 3 mm	ITK		1
Capacitor	Wima	1 μF	ITK		3
Capacitor		10 μF tantalum	ITK		1
Capacitor	Wima	4.7 μF	ITK		1
Capacitor		0.1 μF, SMD	Omega V.		14
Capacitor		0.1 μF, SMD	Omega V.		1
Capacitor		0.22 pF, SMD	Omega V.		2
Reset Switch		Through-hole	Omega V.		1
Crystal OSC		32 HMz	Omega V.		1
Lvl translator	TI	TXS0102DCUR	Digi-Key	296-21931-1-ND	2
Lvl translator	TI	SN74AVC4T774PW	Digi-Key	296-23611-5-ND	2
Connector	TE Connectivity	28pin	Digi-Key	A26493-ND	1
Voltage reg	NS	LM1117 3.3V SOT-223	Digi-Key	LM1117MPX-3.3CT-ND	1
Voltage reg	NS	LP2992 1.8V SOT23-5	Digi-Key	LP2992IM5-1.8CT-ND	1

Table F.1: Expansion board components

Appendix G

Expansion Board Schematics

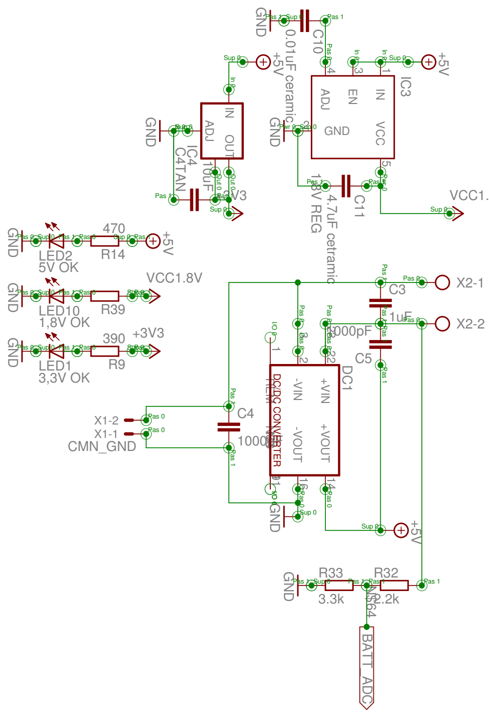


Figure G.1: Expansion board, power supply

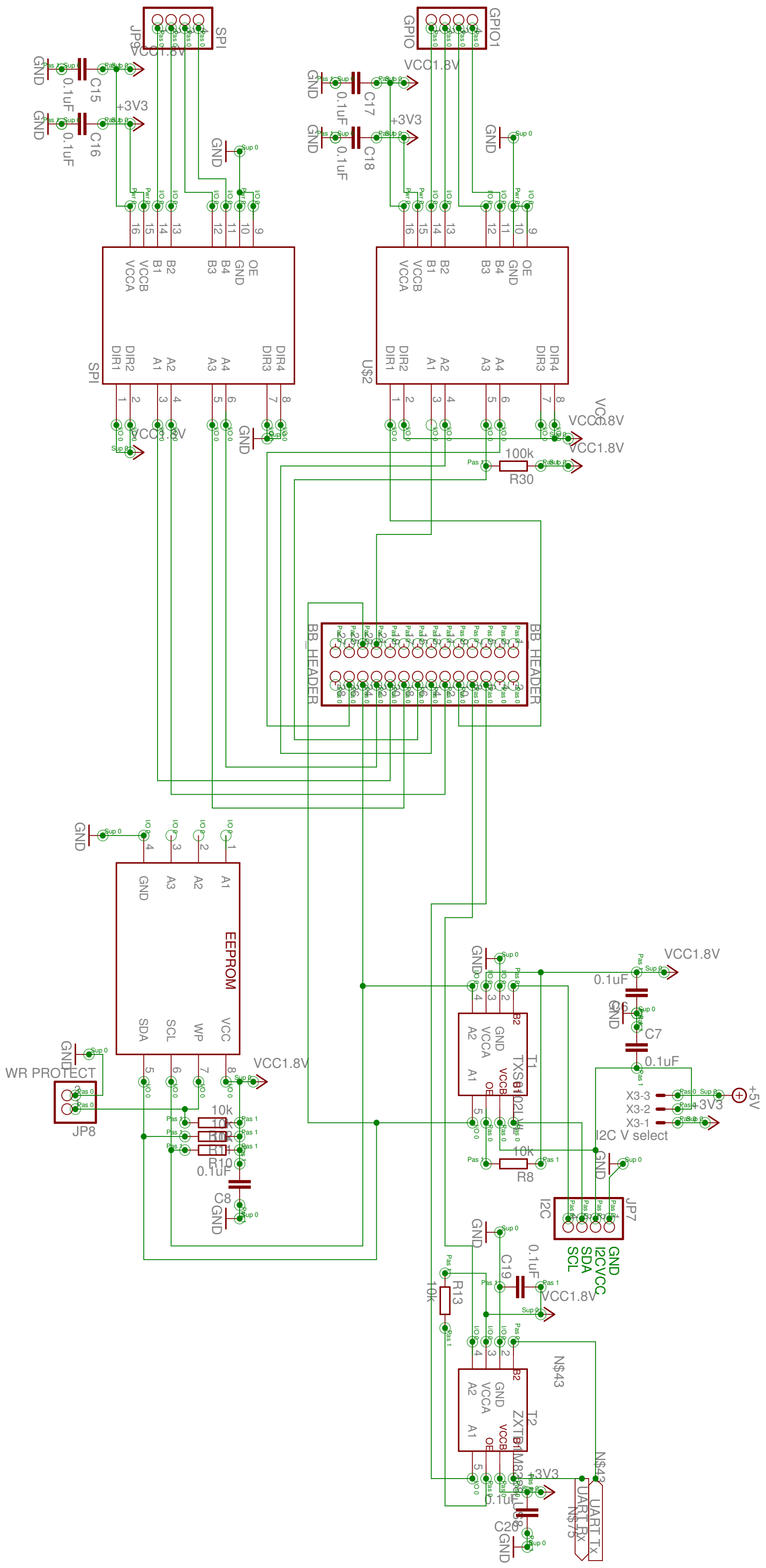


Figure G.2: Expansion board, BeagleBoard interface

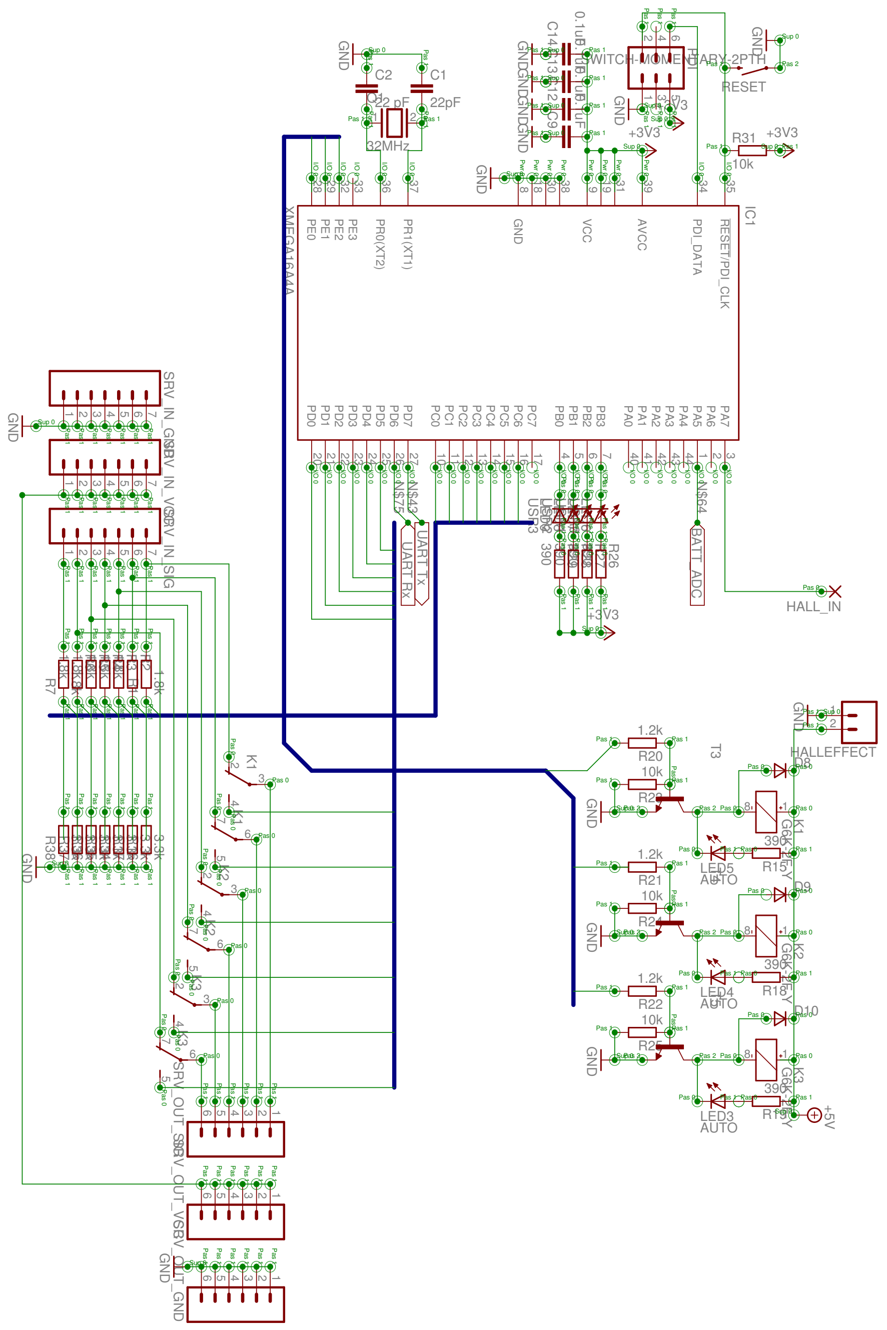


Figure G.3: Expansion board, relays and XMEGA

Appendix H

Expansion Board Development and Test Tools

Item	Producer	Description	Item no/version
Programmer	Atmel	JTAG ICE mkII	rev 10 A09-0041/09
Programmer	Atmel	STK500	AVRATSTK500 ATM012694
Servo	Hitec	HS-322HD	-
Receiver	Orangerx DSM2	-	-
Programming SW	Atmel	AVRStudio 5.0	v 5.0.1038.1038
RC transmitter	Spektrum	DX6i	

Table H.1: Development tools

Appendix I

Programming the Expansion Board EEPROM

Before reading further, one should be aware that the proper EEPROM values for the expansion board are available in Table I.1. The first section of this guide describes how to read from the EEPROM. This is not necessary if one only wish to program a new expansion board that utilizes BB I/O in the same manner as described in the expansion board schematics. (Figure G.2).

- Install i2ctools on the BeagleBoard using `opkg install i2c-tools`. Version 3.0.2 was used in this guide.
- Make sure that `/dev/i2c-2` is listed. In this version of Ångström (kernel 2.6.32, Angstrom 2010.7-test-20100916, built from branch: org.openembedded.dev, revision: 37ac17279e45eeb238d688b68251de8121283490), this is enabled by default and I2C-2 is used on the expansion header.
- For reading, make sure your desired expansion board is attached.
- Run `i2cdetect -r 2` to find all the devices attached to this bus. If the TinCanTools *Trainer* is attached, it should read:

```
 root@beagleboard:~# i2cdetect -r 2
WARNING! This program can confuse your I2C bus, cause data loss↩
    and worse!
I will probe file /dev/i2c-2 using read byte commands.
I will probe address range 0x03-0x77.
Continue? [Y/n]
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: 50 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

This indicates a present device on address 0x50. Should all fields return blank, check expansion power and EEPROM connections.

- To read a register use the command `i2cget (i2cbus chip-address data-address)`. example: `i2cget 2 0x50 0x00` will return the first data field in the EEPROM, which in this case is 0x00:

  ```
  root@beagleboard:/home# i2cget 2 0x50 0
  WARNING! This program can confuse your I2C bus, cause data loss↩
      and worse!
  I will read from device file /dev/i2c-2, chip address 0x50, ↩
     data address
  0x00, using read byte data.
  Continue? [Y/n]
  0x00
  ```

  An unprogrammed field contains 0xff.

  In the case of the trainer, only the fields 0x00 - 0x07 are programmed. The values of these fields are presented in Table I.1.

- To write to a register, first make sure that the “write enable” header is latched. On the expansion board this is the two pins located towards the middle of the board, close to the EEPROM.

- To write to a field, use the command `i2cset (i2cbus chip-address data-address value)`. Example: `i2cset 2 0x50 0x00 0x00` will write 0x00 to the first data field in the EEPROM:

  ```
  root@beagleboard:/etc# i2cset 2 0x50 0x00 0x00
  WARNING! This program can confuse your I2C bus, cause data loss↩
      and worse!
  DANGEROUS! Writing to a serial EEPROM on a memory DIMM
  may render your memory USELESS and make your system UNBOOTABLE!
  I will write to device file /dev/i2c-2, chip address 0x50, data↩
      address
  0x00, data 0x00, mode byte.
  Continue? [y/N]
  ```

- You want to remove the “write enable” header when finished and use i2cget to verify the contents.

- Upon BB reboot it should now output over the serial line:

  ```
  Texas Instruments X-Loader 1.4.4ss (Sep 10 2010 - 02:43:02)
  Beagle Rev C4
  .
  .
  .
  Probing for expansion boards, if none are connected you ll see ↩
     a harmless I2C error.


  Recognized Tincantools Trainer expansion board (rev 0 0)
  ```

```
Beagle Rev C4
.
.
.
```

The pinmux should now be set appropriately, enabling expansion board communication over USART, SPI, I^{2}C and GPIO.

i2cbus	chip address	data address	value
2	0x50	0x00	0x00
2	0x50	0x01	0x01
2	0x50	0x02	0x00
2	0x50	0x03	0x04
2	0x50	0x04	0x00
2	0x50	0x05	0x00
2	0x50	0x06	0x30
2	0x50	0x07	0x00
2	0x50	0x08	0xff
2	0x50	0x09	0xff
2	0x50	0x...	0xff

Table I.1: Expansion board EEPROM values

Appendix J

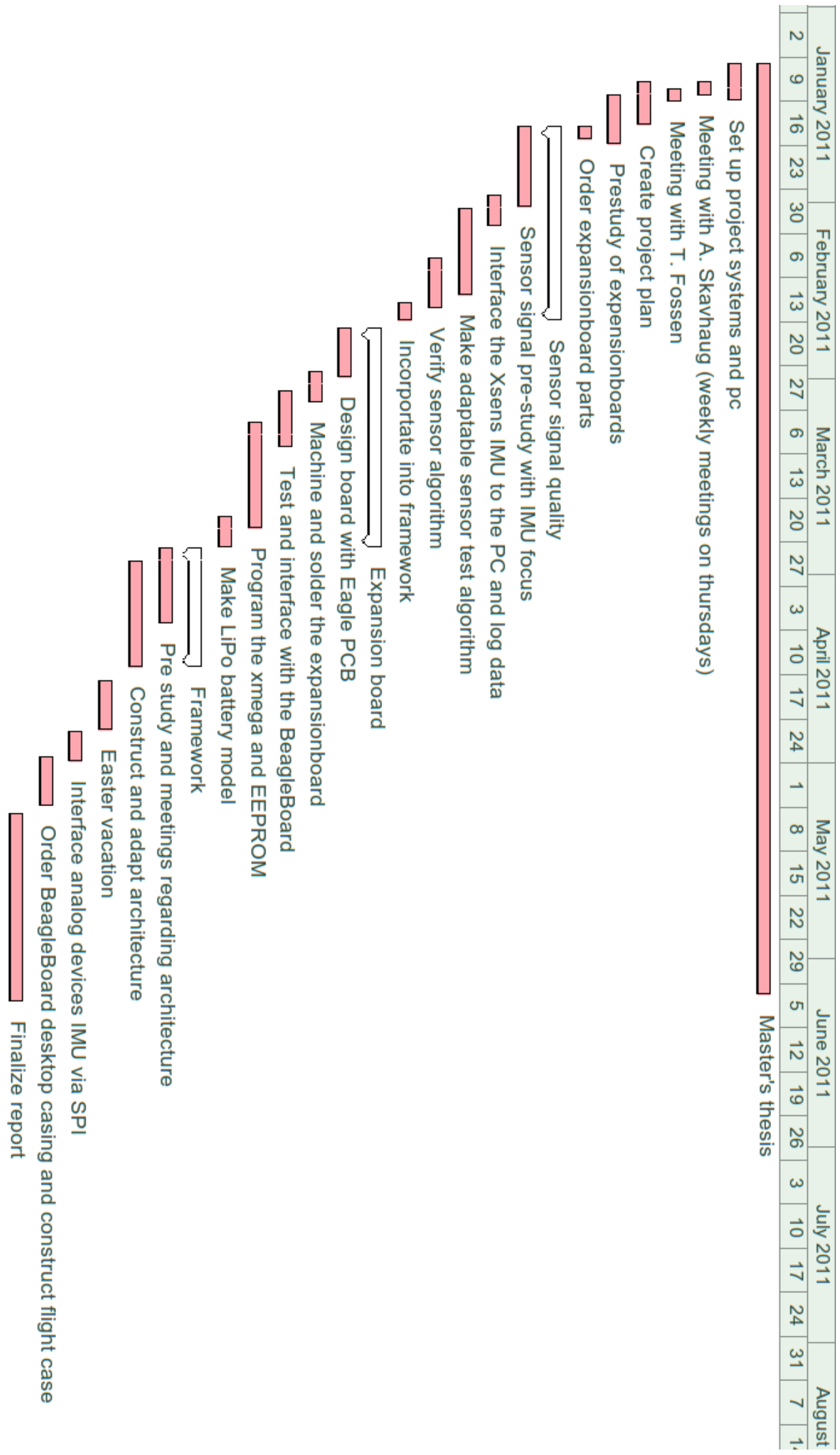
Project Gantt Charts

This appendix contains the Gannt charts which illustrates the planned progress. The project plan was revised monthly. Updated charts for each month with comments can be viewed here.

J.1 January Revision

This is the initial plan, and will be revised throughout the project.

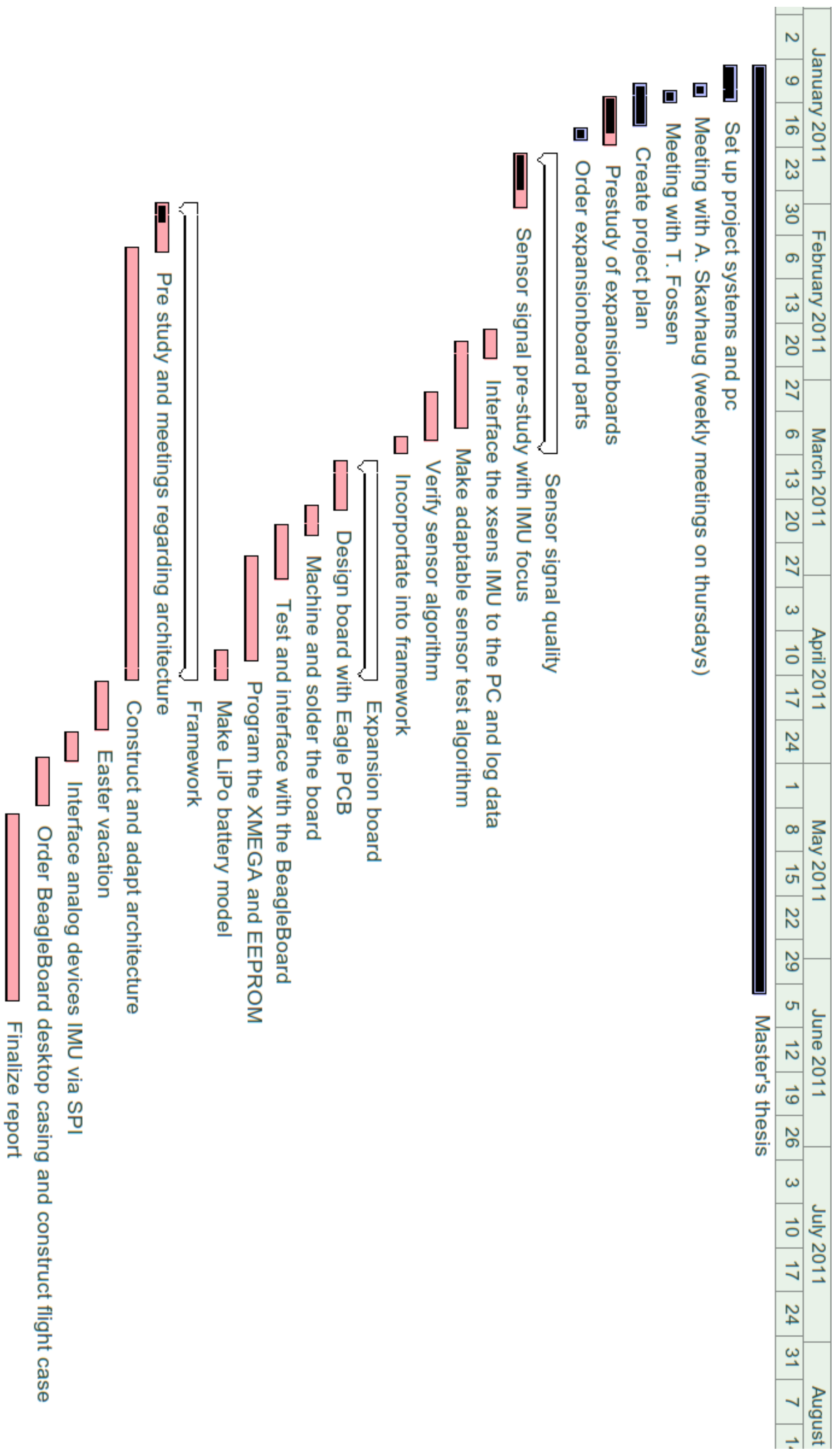
- The project has been divided into three main parts: Sensor error detection, expansion board and framework.
- Some minor, less important tasks have been added towards the end.
- Initial meetings are done 10-14 Jan, and the thesis work begins Jan 17th.


January 2011
February 2011
March 2011
April 2011
May 2011
June 2011
July 2011
August
2 9 16 23 30 6 13 20 27 6 13 20 27 3 10 17 24 1 8 15 22 29 5 12 19 26 3 10 17 24 31 7
Master's thesis
Set up project systems and pc
Meeting with A. Skavhaug (weekly meetings on thursdays)
Meeting with T. Fossen
Create project plan
Prestudy of expensionboards
Order expansionboard parts
Sensor signal quality
Sensor signal pre-study with IMU focus
Interface the Xsens IMU to the PC and log data
Make adaptable sensor test algorithm
Verify sensor algorithm
Incorportate into framework
Expansion board
Design board with Eagle PCB
Machine and solder the expansionboard
Test and interface with the BeagleBoard
Program the xmega and EEPROM
Make LiPo battery model
Framework
Pre study and meetings regarding architecture
Construct and adapt architecture
Easter vacation
Interface analog devices IMU via SPI
Order BeagleBoard desktop casing and construct flight case
Finalize report

J.2 February Revision

This revison was made February 2nd.

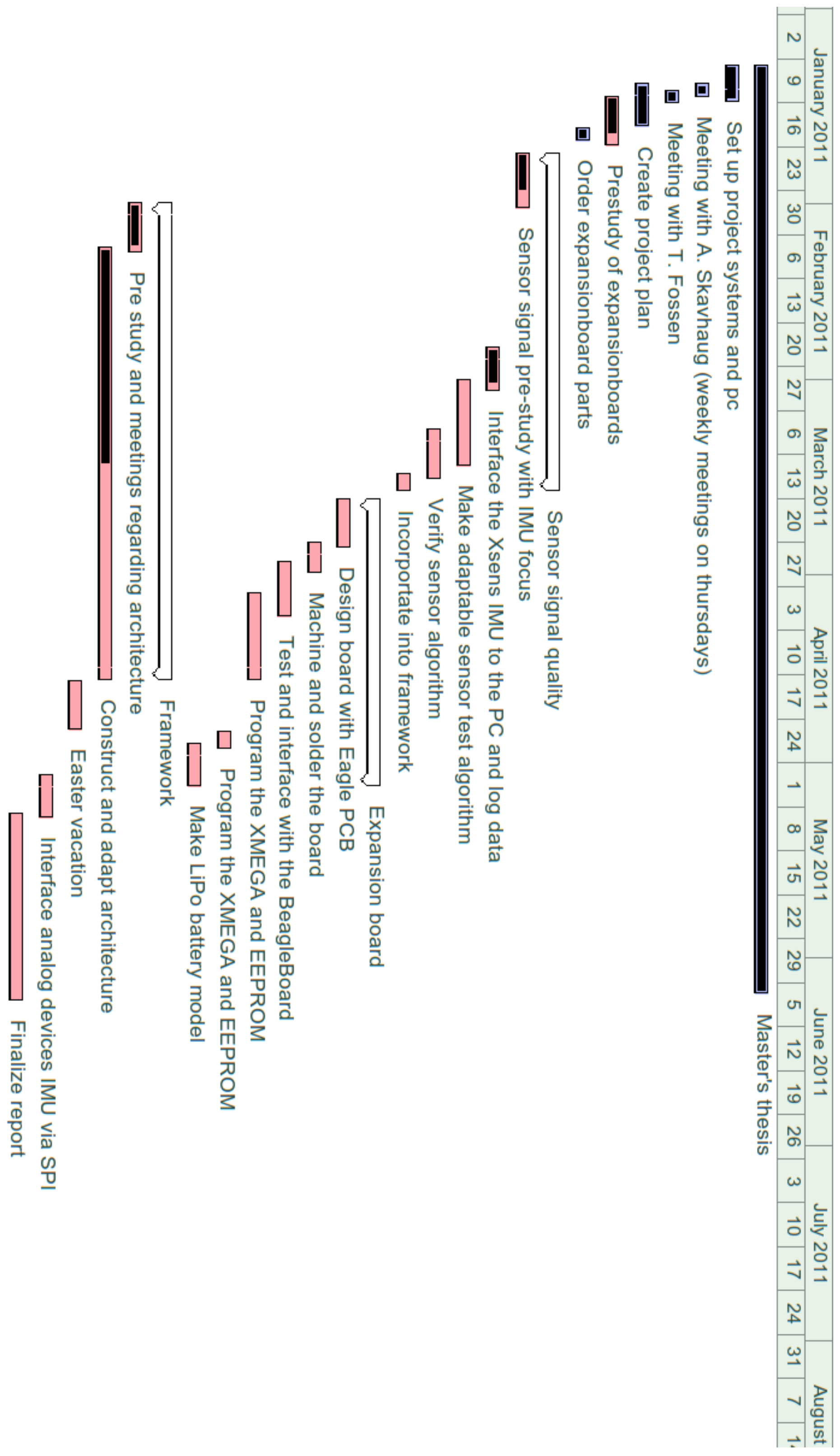
- In the first weekly meeting, A. Skavhaug pointed out that the framework should always be kept in mind. Alterations and needed adjustments will show as the project progresses. The framework pre-study and implementation was moved forward, and implementation showed as a long process. The basics will be implemented, and expansions made throughout the project.
- The remaining tasks were moved further ahead to accommodate the point above.


January 2011
February 2011
March 2011
April 2011
May 2011
June 2011
July 2011
August
2
9
16
23
30
6
13
20
27
6
13
20
27
3
10
17
24
1
8
15
22
29
5
12
19
26
3
10
17
24
31
7
Master's thesis
Set up project systems and pc
Meeting with A. Skavhaug (weekly meetings on thursdays)
Meeting with T. Fossen
Create project plan
Prestudy of expansionboards
Order expansionboard parts
Sensor signal quality
Sensor signal pre-study with IMU focus
Interface the xsens IMU to the PC and log data
Make adaptable sensor test algorithm
Verify sensor algorithm
Incorportate into framework
Expansion board
Design board with Eagle PCB
Machine and solder the board
Test and interface with the BeagleBoard
Program the XMEGA and EEPROM
Make LiPo battery model
Framework
Pre study and meetings regarding architecture
Construct and adapt architecture
Easter vacation
Interface analog devices IMU via SPI
Order BeagleBoard desktop casing and construct flight case
Finalize report

J.3 March Revision

This revision was made March 3rd.

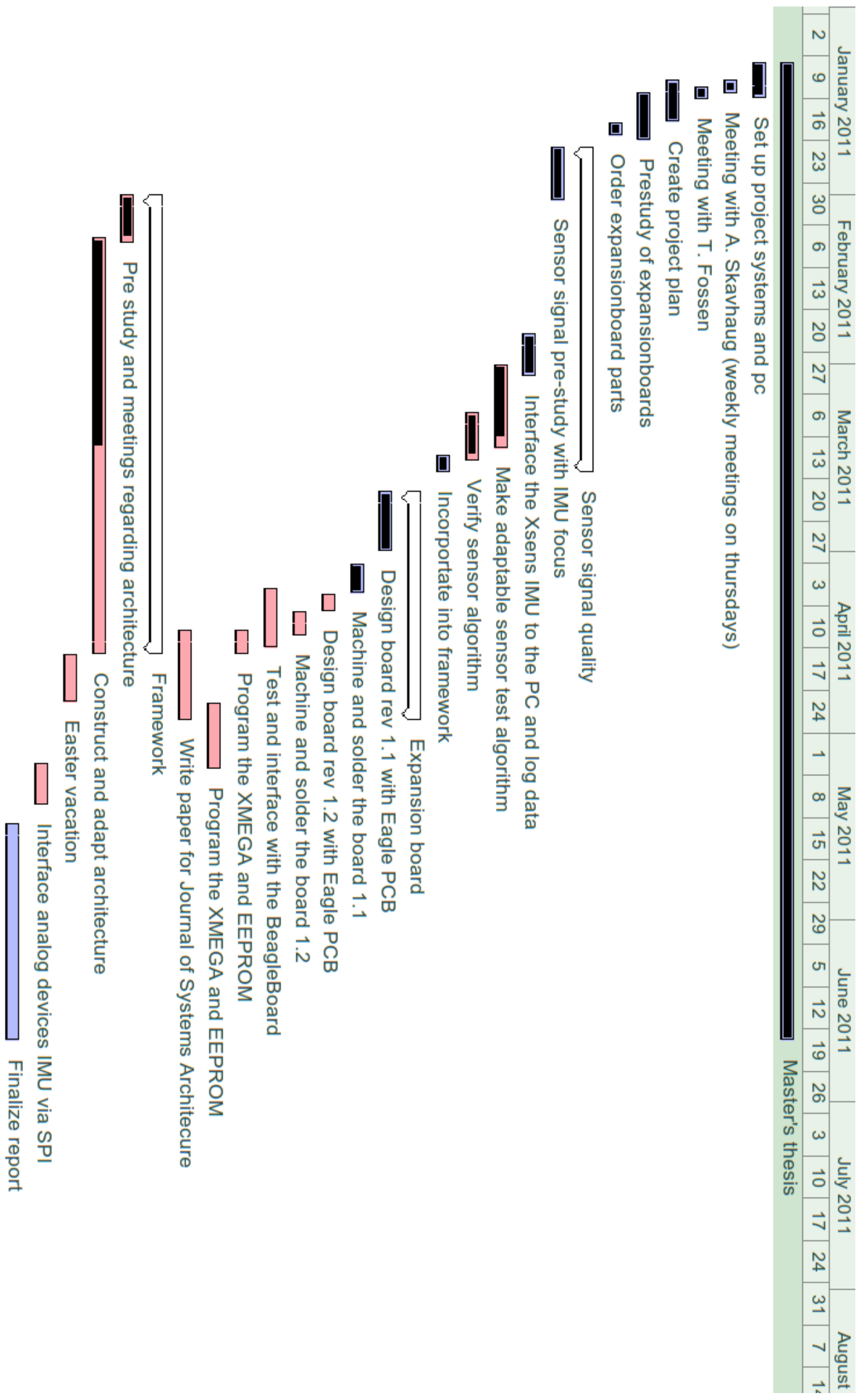
- The framework proved to be a larger task than anticipated. To create a fully functional framework took half a week longer than planned, stretching this part of the project to 2.5 weeks.
- Due to the above, and the fact that some tasks surrounding the framework must be done alongside other tasks, the BeagleBoard casing is phased out as part of the plan.


January 2011
February 2011
March 2011
April 2011
May 2011
June 2011
July 2011
August
2
9
16
23
30
6
13
20
27
6
13
20
27
3
10
17
24
1
8
15
22
29
5
12
19
26
3
10
17
24
31
7
Master's thesis
Set up project systems and pc
Meeting with A. Skavhaug (weekly meetings on thursdays)
Meeting with T. Fossen
Create project plan
Prestudy of expansionboards
Order expansionboard parts
Sensor signal quality
Sensor signal pre-study with IMU focus
Interface the Xsens IMU to the PC and log data
Make adaptable sensor test algorithm
Verify sensor algorithm
Incorporate into framework
Expansion board
Design board with Eagle PCB
Machine and solder the board
Test and interface with the BeagleBoard
Program the XMEGA and EEPROM
Program the XMEGA and EEPROM
Make LiPo battery model
Framework
Pre study and meetings regarding architecture
Construct and adapt architecture
Easter vacation
Interface analog devices IMU via SPI
Finalize report

J.4 April Revision

This revision was made April 10th.

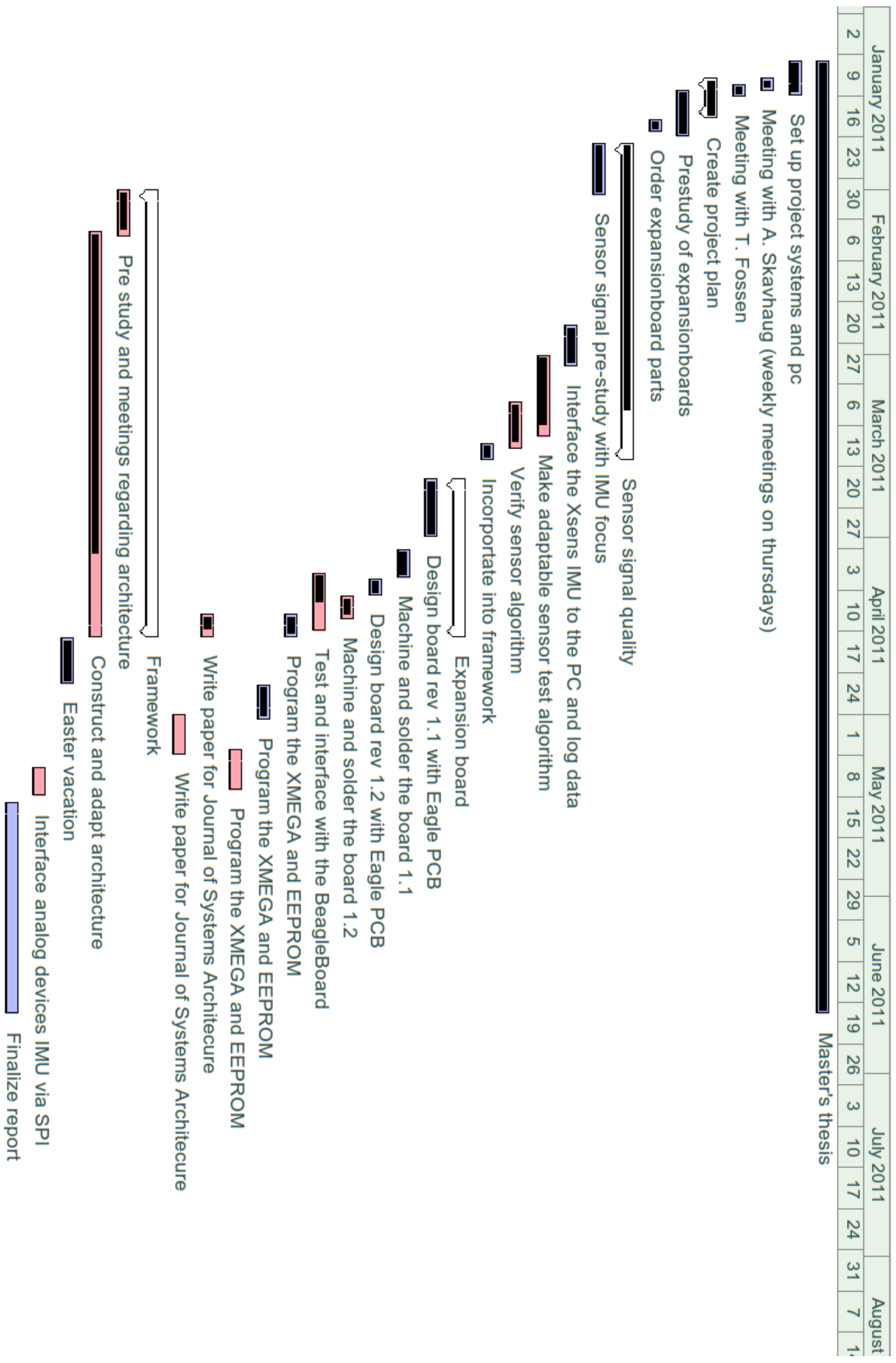
- Together with Amund Skavhaug it was decided to publish a paper regarding the SW framework and publish in Journal of Systems Architecture. One week extra has been reserved for this purpose. Note that the thesis deadline has been moved one week forward.
- The complexity combined with the limited amount of space on the expansion board required more design focus than what was anticipated. The design process took approximately 5 days longer than planned.
- Due to Digi-Key sending the wrong component (a txs0102 level converter was ordered and some other unknown component shipped) the debugging process was very extensive. The shipped item was almost the same package as the txs0102, and the label on the package stated that this indeed was the correct device. An isolated test confirmed that is was the wrong device. The expansion board was in such a bad state after this debugging that a new board had to be designed, milled and soldered. Fortunately this was a chance to get some other minor redesigns done. This newest and final version of the board is known as version 1.2
- Due to these events the LiPo battery model has been phased out of the schedule. A battery model was made during the autumn which might be easily integrated.


January 2011
February 2011
March 2011
April 2011
May 2011
June 2011
July 2011
August
2 9 16 23 30 6 13 20 27 6 13 20 27 3 10 17 24 1 8 15 22 29 5 12 19 26 3 10 17 24 31 7
Master's thesis
Set up project systems and pc
Meeting with A. Skavhaug (weekly meetings on thursdays)
Meeting with T. Fossen
Create project plan
Prestudy of expansionboards
Order expansionboard parts
Sensor signal quality
Sensor signal pre-study with IMU focus
Interface the Xsens IMU to the PC and log data
Make adaptable sensor test algorithm
Verify sensor algorithm
Incorportate into framework
Expansion board
Design board rev 1.1 with Eagle PCB
Machine and solder the board 1.1
Design board rev 1.2 with Eagle PCB
Machine and solder the board 1.2
Test and interface with the BeagleBoard
Program the XMEGA and EEPROM
Program the XMEGA and EEPROM
Framework
Write paper for Journal of Systems Architecure
Pre study and meetings regarding architecture
Construct and adapt architecture
Easter vacation
Interface analog devices IMU via SPI
Finalize report

J.5 May Revision

This revision was made May 3rd.

- After commencing with the paper, it is apparent that quite some timing and/or profiling data is needed to document the results. This will take some time, but hopefully bring both the paper and the report to a higher level.
- The new txs0102 chips from Farnell had not arrived. Some time must be reserved to solder and test the BB - expansion board communication.
- It must be evaluated if it is feasible to connect the Analog Devices IMU via SPI in time.


January 2011
February 2011
March 2011
April 2011
May 2011
June 2011
July 2011
August
2 9 16 23 30 6 13 20 27 6 13 20 27 3 10 17 24 1 8 15 22 29 5 12 19 26 3 10 17 24 31 7
Master's thesis
Set up project systems and pc
Meeting with A. Skavhaug (weekly meetings on thursdays)
Meeting with T. Fossen
Create project plan
Prestudy of expansionboards
Order expansionboard parts
Sensor signal quality
Sensor signal pre-study with IMU focus
Interface the Xsens IMU to the PC and log data
Make adaptable sensor test algorithm
Verify sensor algorithm
Incorportate into framework
Expansion board
Design board rev 1.1 with Eagle PCB
Machine and solder the board 1.1
Design board rev 1.2 with Eagle PCB
Machine and solder the board 1.2
Test and interface with the BeagleBoard
Program the XMEGA and EEPROM
Program the XMEGA and EEPROM
Program the XMEGA and EEPROM
Write paper for Journal of Systems Architecure
Write paper for Journal of Systems Architecure
Framework
Pre study and meetings regarding architecture
Construct and adapt architecture
Easter vacation
Interface analog devices IMU via SPI
Finalize report