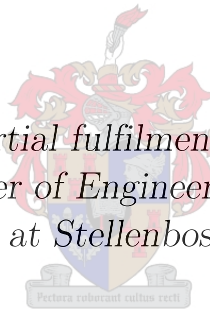
Specification, Design and Implementation of a Flight Control Unit for an Unmanned Aerial Vehicle

by

Hartmut Behrens



Thesis presented in partial fulfilment of the requirements for the degree of Master of Engineering in the Faculty of Engineering at Stellenbosch University

Department of Electrical and Electronic Engineering,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Supervisor: Prof T. Jones

December 2015

Stellenbosch University https://scholar.sun.ac.za

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: ....

Copyright © 2015 Stellenbosch University
All rights reserved.

Stellenbosch University https://scholar.sun.ac.za

Abstract

The specification, design and implementation of an avionics system including a flight control unit for an unmanned aerial vehicle that is suitable for research purposes is presented. The project aims to address a number of limitations of previous systems that were developed at the Electronic Systems Laboratory of Stellenbosch University.

An architecture is developed that is based on documented research and open industry standards. The architecture enables distributed, real-time and embedded flight control applications to be developed. The platform is extendible with minimal effort. Provision is made to support sensor and actuator hardware that has been developed in the past.

The results of a flight controller that was previously developed at the Electronic Systems Laboratory and was ported to the new architecture are provided. The newly ported flight controller is proven to work correctly using hardware-in-the-loop simulations. Comparative tests indicate that the performance is on-par with existing Electronic Systems Laboratory systems.

Stellenbosch University https://scholar.sun.ac.za

Opsomming

Die spesifikasie, ontwerp en implementering van avionika, insluitende 'n vlugbeheerstelsel vir 'n onbemande vliegtuig wat geskik is vir navorsing doeleindes, word aangebied. Die projek spreek 'n aantal beperkinge aan van vorige stelsels wat by die Elektroniese Stelses Laboratorium van die Universiteit van Stellenbosch ontwikkel is.

'n Argitektuur is ontwikkel wat gebaseer is op vorige navorsing en oop industrie standaarde. Die argitektuur maak dit moontlik om 'n verspreide, geïntegreerde en intydse vlugbeheerstelsel te ontwikkel. Die platform kan met minimale inspanning opgegradeer word. Voorsiening is ook gemaak om bestaande sensors en aktueerders te ondersteun.

Die resultate van 'n bestaande vlugbeheerstelsel wat oorgedra is na die nuwe argitektuur word beskryf. Dit word gewys dat die nuwe stelsel korrek werk met behulp van hardeware-in-die-lus simulasies. Vergelykende toetse dui daarop dat die prestasie op gelyke voet is met bestaande stelsels van die Elektroniese Stelses Laboratorium.

Stellenbosch University https://scholar.sun.ac.za

Acknowledgements

I would like to express my sincere gratitude to:

- Prof T. Jones for affording me the opportunity to complete this project in the ESL and guiding me in the right direction during our discussions.
- My Wife and Family - Thank you - for too many reasons to list here !

Stellenbosch University https://scholar.sun.ac.za

Contents

Declaration **i**

Abstract **ii**

Opsomming **iii**

Acknowledgements **iv**

Contents **v**

List of Figures **vii**

Nomenclature **x**

1 Flight Control Systems **1**
1.1 Introduction 1
1.2 Limitations 2
1.3 Objectives / Requirements 4
1.4 Evaluation 5
1.5 Approach 5
1.6 Thesis Overview 5

2 System Design **6**
2.1 Types 6
2.2 Distributed Systems 8
2.3 Software 9
2.4 Components 12
2.5 Standards 13
2.6 Model Driven Development 13
2.7 Real-time Operating System 14
2.8 Compatibility 15
2.9 Conclusion 15

3 Architecture **17**
3.1 Structure 17

Stellenbosch University https://scholar.sun.ac.za

3.2 Components . . . 19
3.3 Event Service . . . 22
3.4 Flow Policies . . . 22
3.5 Distributed Systems . . . 27
3.6 Hardware Block Diagram . . . 29
3.7 Software Block Diagram . . . 30
3.8 Simulation . . . 32
3.9 CAN-Ethernet Gateway . . . 33
3.10 Conclusion . . . 33

4 Development **35**
4.1 Hardware . . . 35
4.2 Real-time Operating System . . . 37
4.3 Software . . . 38
4.4 Conclusion . . . 48

5 System Integration **49**
5.1 Hardware . . . 49
5.2 Software . . . 50
5.3 Conclusion . . . 56

6 System Verification **57**
6.1 CAN-Ethernet Gateway . . . 58
6.2 Real-time Operation . . . 59
6.3 Ethernet Switch . . . 62
6.4 Comparison . . . 62
6.5 Flight Control System . . . 77
6.6 Distributed Flight Control System . . . 82
6.7 Conclusion . . . 87

7 Summary and Recommendations **89**
7.1 Summary . . . 89
7.2 Recommendations . . . 89

Appendices **91**

A Software **92**
A.1 Serial Communication Protocol . . . 92

List of References **93**

Stellenbosch University https://scholar.sun.ac.za

List of Figures

1.1 Overview of an ESL flight control system. . . . 2

2.1 Independent avionics [1]. . . . 6
2.2 Federated avionics [1]. . . . 7
2.3 Integrated modular avionics [1]. . . . 8
2.4 Middleware layers [2]. . . . 12

3.1 Template Flight Control System Architecture [3]. . . . 18
3.2 Ports and attributes of a software component 20
3.3 IDL to C++ mapping 21
3.4 A basic flight control system 21
3.5 Publish/Subscribe architecture using an event service. . . . 22
3.6 Flow Chart of periodic tasks [4]. . . . 25
3.7 Control and data flow diagram. . . . 25
3.8 Remote method invocation. . . . 28
3.9 Hardware block diagram of the flight control system. . . . 29
3.10 Component diagram of the flight control system. . . . 30
3.11 HIL simulation of existing FCS [11]. . . . 32

4.1 Rate Generator component interface. . . . 38
4.2 Command component interface. . . . 40
4.3 HILInterface component interface. . . . 40
4.4 GPS component interface. . . . 42
4.5 IMU component interface. . . . 43
4.6 Estimator component interface. . . . 44
4.7 Controller component interface. . . . 45
4.8 Servo component interface. . . . 46
4.9 OnBoardComputer component interface. . . . 47
4.10 Existing HIL simulation setup. . . . 48
4.11 New HIL simulation setup. . . . 48

5.1 HIL tested hardware block diagram of the flight control system. . . . 50
5.2 Rate Generator component interface. . . . 50
5.3 Visual modelling of port connection. . . . 55

Stellenbosch University https://scholar.sun.ac.za

6.1 CAN-Ethernet gateway test setup. 58
6.2 Event latency. 59
6.3 Latency distribution graph. 60
6.4 Total algorithm execution time. 61
6.5 Execution event interval. 61
6.6 Execution event interval after adjustment. 62
6.7 Setup for simultaneous operation of both flight control systems. . . 63
6.8 Way-points that were used during the simulation. 64
6.9 Latitude comparison. 66
6.10 Longitude comparison. 66
6.11 GPS north velocity comparison. 67
6.12 GPS east velocity comparison. 67
6.13 Gyro x-axis/roll angle comparison. 68
6.14 Gyro y-axis/pitch angle comparison. 68
6.15 Gyro z-axis/yaw angle comparison. 69
6.16 Pitot-static indicated airspeed. 69
6.17 Pitot-static pressure altitude. 70
6.18 Estimator north position comparison. 70
6.19 Estimator east position comparison. 71
6.20 Estimator north velocity comparison. 71
6.21 Estimator east velocity comparison. 72
6.22 Estimator roll angle comparison. 72
6.23 Estimator pitch angle comparison. 73
6.24 Estimator yaw angle comparison. 73
6.25 Controller bank angle reference comparison. 74
6.26 Controller horizontal acceleration reference comparison. 74
6.27 Comparison of navigated waypoints. 75
6.28 Comparison of actuator throttle commands. 75
6.29 Comparison of left aileron commands. 76
6.30 Comparison of actuator elevator commands. 76
6.31 Comparison of actuator rudder commands. 77
6.32 Next destination way-point selected by controller. 78
6.33 GPS Longitude vs Latitude. 78
6.34 Pitot-static pressure altitude. 79
6.35 Estimator determined Longitude vs Latitude. 79
6.36 Pitot-static indicated airspeed. 80
6.37 Actuator throttle command. 80
6.38 Actuator elevator command. 81
6.39 Actuator rudder command. 81
6.40 Actuator left aileron command. 82
6.41 Waypoints for the distributed test. 83
6.42 GPS Longitude vs Latitude. 84
6.43 Pitot-static pressure altitude. 84
6.44 Estimator determined Longitude vs Latitude. 84

Stellenbosch University https://scholar.sun.ac.za

6.45 Pitot-static indicated airspeed. 85
6.46 Actuator throttle command. 85
6.47 Actuator elevator command. 86
6.48 Actuator rudder command. 86
6.49 Actuator left aileron command. 87

Stellenbosch University https://scholar.sun.ac.za

Nomenclature

Acronyms

AFDX Avionics Full Duplex Ethernet

ARINC Aeronautical Radio Incorporated

ARM Advanced RISC Machines

CAN Controller Area Network

CORBA Common Object Request Broker Architecture

COTS Commercial Off The Shelf

DRE Distributed, Real-Time, Embedded

ESL Electronic Systems Laboratory

FCS Flight Control System

FPGA Field Programmable Gate Array

FOSS Free Open Source Software

GME Generic Modelling Environment

GPS Global Positioning System

HIL Hardware In the Loop

IDL Interface Description Language

IMA Integrated Modular Avionics

Stellenbosch University https://scholar.sun.ac.za

IMU Inertial Measurement Unit

IOR Interoperable Object Reference

IP Internet Protocol

LwCCM Lightweight CORBA Component Model

LRU Line Replaceable Unit

MVC Model-View-Controller

OMG Object Management Group

OO Object Oriented

ORB Object Request Broker

OS Operating System

RAM Random Access Memory

RC Radio Controlled

RTOS Real-Time Operating System

SBC Single Board Computer

UAV Unmanned Aerial Vehicle

UCM Unified Component Model

UDP User Datagram Protocol

XML Extensible Markup Language

Stellenbosch University https://scholar.sun.ac.za

Chapter 1

Flight Control Systems

1.1 Introduction

The Electronic Systems Laboratory (ESL) at Stellenbosch University is a centre of expertise in the modelling, control and automation of aerospace, terrestrial and underwater vehicles. As a result of research activities, a number of flight control platforms have already been developed at the ESL that addressed various flight control problems related to autonomous flight. The platforms that were designed as a result of the conducted research activities provided an efficient solution to the control problem that was being studied.

The current state of avionics that include flight control systems in the ESL can be briefly summarized as follows: The first project at the ESL that automated the flight of a fixed wing aircraft produced a micro-controller based flight avionics package with a Controller Area Network (CAN) fieldbus [5]. The package was refined by adding hardware-in-the-loop simulation functionality during research that implemented autonomous take-off and landing capability for the fixed wing unmanned aerial vehicle (UAV) [6]. At the same time, an alternative flight control platform that off-loaded most of the processing to a ground station via a high speed link was also developed for an electrically powered radio-controlled (RC) helicopter [7]. Also during the same year, an avionics package that was based on a PC/104 motherboard equipped with a CAN fieldbus extension running a Linux-based operating system (OS) was produced [8] [9]. It was initially used as a rotary wing testbed and also to perform research on vertical take-off and landing (VTOL). The PC/104 setup was later improved as part of research aimed at introducing aerobatic flight manoeuvrability during autonomous flight [10]. A light weight, low-power drop in replacement for the PC/104 avionics was eventually designed during research on a variable stability UAV. This variant however lacked the ability to run a Linux-based OS [11]. It is the platform that is currently predominantly used at the ESL.

Stellenbosch University https://scholar.sun.ac.za

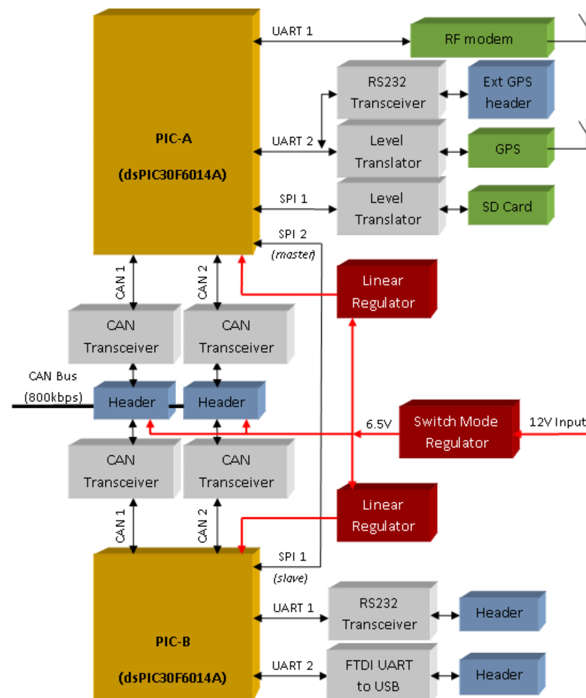


Figure 1.1: Overview of an ESL flight control system.

Through all these studies, extensive experience has been gained in developing complex control algorithms that equip UAVs with new autonomous flight capabilities. The efficient implementation of systems realizing these capabilities has in part been made possible by the introduction of digital computing hardware into the control loop [12]. However, architectural considerations such as maintainability and upgradability of the avionics were often secondary concerns that were relegated in favour of demonstrating the achievement of research objectives.

1.2 Limitations

The avionics systems that have been developed at the ESL are purpose-built for the task at hand, with the architecture typically being dictated by the flight control algorithms being implemented. The systems consist of a centralized processor with tightly coupled subsystems, each partitioned to perform a specific function, that are bound together into a single unit performing the required aircraft control. Each subsystem is developed from scratch with minimal technology reuse. Data is shared either via specific analogue/digital signals or via a low-speed fieldbus such as CAN. Sensors and actuators are also connected via the CAN fieldbus. Figure 1.1 illustrates the architecture of an existing flight control system at the ESL.

Stellenbosch University https://scholar.sun.ac.za

While this custom architecture yields a highly efficient design - it is at the same time also inflexible, since it is built and wired specifically for the task at hand. Consider, for example, the emerging trend of vision-based navigation, a first study of which has been completed at Stellenbosch University [13]. It was difficult to implement a flight-capable demonstrator model for the study using only the avionics available at that time due to the high processing power and bandwidth requirements of the vision-based control algorithm. To overcome the limitation, dedicated field programmable gate array (FPGA) based hardware was developed for the project that performed most of the image processing functions. As this example illustrates, to keep up with an expected increase in capability, an extension or redevelopment of the avionics would often be needed every time a new capability is required.

The use of embedded hardware to implement control systems has also introduced software (firmware) into the control loop which must be programmed, supported and eventually evolved [12] . Since software development is error-prone, mechanisms are required that will allow the developer to manage the complexity and reduce the occurrence of errors. A mechanism, which has already been implemented and used effectively at the ESL, is a hardware-in-the-loop (HIL) simulator which allows the control systems to be tested and proven under conditions resembling those encountered in the field. Another mechanism that could achieve a reduction in encountered errors is to rely more on previously developed, proven and tested software. Unfortunately this is not easily done on a custom system, since the software is tied to a great degree to the underlying hardware.

Beyond this, flight control software needs to periodically process tasks in a deterministic way with real-time deadlines. The correctness of an algorithmic computation depends not only on the logical correctness of the calculated value but also on the timely delivery of the result. Should this not occur, the consequences could be severe - in the worst case, flight control could be lost.

Enforcing real-time constraints with software developed from scratch could result in a sub-optimal implementation. Developing software that meets real-time requirements often also results in an implementation that is tightly coupled to the underlying hardware. The inflexibility of tightly coupled software can lead to more expensive development effort when changes are required due to e.g. the availability of new / obsolescence of old hardware . Additionally, tight coupling makes reuse of existing, tested software components substantially more difficult.

Stellenbosch University https://scholar.sun.ac.za

The limitations of the current ESL avionics systems can therefore be summarized as follows:

- The system has limited processor, memory and fieldbus bandwidth resources.
- Upgrading system resources often requires a redesign.
- The developed software is tightly coupled to the underlying hardware.
- The scheduling is hard-coded in the main cycle, with poor ability to support asynchronous timing requirements.
- The scheduling assumes all tasks need to run at the same rate of repetition with the same priority.
- Application logic is evolved around the execution ordering hard-coded in the main cycle.
- It could be difficult to accommodate multi-threaded operations with the current scheduling technique.
- It is difficult to extend the system with new payload hardware that may be required for specific research purposes - e.g. a camera subsystem for vision-related flight control research.

1.3 Objectives / Requirements

The project aims to specify, design and implement a new avionics system that is suitable for autonomous flight research purposes while also addressing the identified limitations of the previously developed systems. The objectives of the project will be to :

- develop a flight control system that is modern.
- address the existing system resource limitations.
- make the system and its resources extendible.
- enable re-use of developed software and technology.
- develop a system that is based on open industry standards.
- take into account the real-time, embedded nature of the flight control system.
- make provision to incorporate existing sensors and actuators.

Stellenbosch University https://scholar.sun.ac.za

- develop a system that is suitable for research purposes.
- port an existing flight control system to the new architecture as a proof of concept exercise.

1.4 Evaluation

In order to quantitatively evaluate whether the objectives have been met, the flight control logic of an existing flight control system will be ported to the new hardware / software architecture. Its performance will be evaluated against the existing system from which it was ported using Hardware-in-the-Loop (HIL) simulation results.

1.5 Approach

The approach taken by the thesis to realize the objectives of the study was to investigate whether advances in computer hardware, software and sensor technologies that are relevant to the domain provided promising cues for the development of a modern, re-usable and extendible platform. While a majority of research in autonomous flight is aimed at solving interesting flight control problems - with the usual associated flight controller design to test the validity of study findings - projects focused on the architecture of the platform do also exist. Academic research results as well as industry trends were investigated for their suitability to overcome the limitations of the current architecture.

1.6 Thesis Overview

In chapter 2 the alternatives to meet the project objectives are introduced and a system design is sketched out. In chapter 3 the architecture and functional requirements of the avionics system are developed. The individual hardware and software modules of the new avionics system are designed in detail in chapter 4. This chapter includes the porting and design of an existing ESL flight control system. In chapter 5 the process of combining the individually designed modules into a functioning flight control system is described, while the results of testing and verifying that the completed system works as intended using a HIL simulation are described in chapter 6. The thesis concludes in chapter 7 with recommendations on future developments.

Stellenbosch University https://scholar.sun.ac.za

Chapter 2

System Design

Chapter 1 gave an overview of the current state of flight control hardware and software available in the ESL. It described the architectural limitations that currently exist with these systems. The objectives for a new system that will address the current limitations were also provided. In this chapter, suitable approaches which can be employed to address each of the objectives are introduced and evaluated.

2.1 Types

Avionics systems can generally be divided into three architectural paradigms: independent (analogue), federated and integrated modular architectures [1]. The first generation of avionics architectures were also known as independent avionics. Each function (e.g. flight control, navigation, communication) had its own separate, dedicated sensors and processors. Where an interconnect was required it was achieved using point-to-point wiring. A majority of independent avionics consisted of analogue systems. Figure 2.1 represents an overview of such a system.

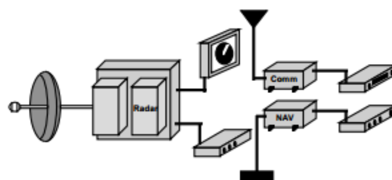


Figure 2.1: Independent avionics [1].

Federated avionics architectures distribute the individual functions of the system over dedicated modules, known as line replaceable units (LRU), each

Stellenbosch University https://scholar.sun.ac.za

of which contains its own dedicated processing resources. An LRU is equipped with specifically developed hardware and software that is suited to the function it needs to perform. Each LRU has few dependencies on other LRU's in the system. The LRU's are connected via a communication bus to form a complete avionics system. Since each LRU is contained in a separate hardware box, a major advantage of federated avionics system is that it is fault tolerant. However, a disadvantage of this approach is that the introduction of a new function into the system also requires the development of a new LRU and thus also of new hardware and software. Figure 2.2 illustrates this concept.

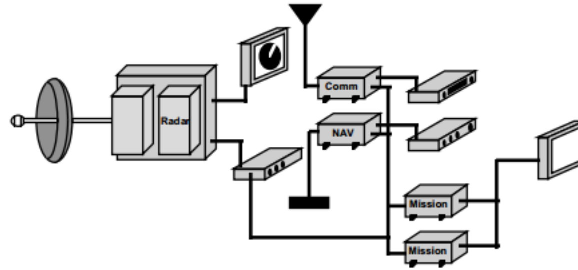


Figure 2.2: Federated avionics [1].

A current trend in the aerospace industry is to develop airborne systems in a modular manner according to the Integrated Modular Avionics (IMA) concept. IMA architecture describes a distributed, real-time computer network that could consist of numerous modules, each of which performing functions that may have different levels of safety-criticality associated with it [14]. Each module could have several partitions that run on top of a real-time OS. Each partition has shared system resources like CPU and memory available to it, but also includes protection mechanisms that allow it to function regardless of failures on other partitions. The Aeronautical Radio Incorporated (ARINC) subsidiary of Rockwell Collins is the major supplier of specifications for the IMA architecture: ARINC 650 and ARINC 651 describe general hardware and software standards for the IMA architecture while ARINC 653 is the software specification describing the partitioning mechanism that needs to be implemented on the underlying (real-time) operating system. ARINC specifications are voluntary aviation technical standards that are available for purchase. Although IMA architectures are most often associated with large aircraft, promising Linux-based research implementations for small UAVs have also been built [15]. Figure 2.3 provides a top-level overview of an IMA-based avionics system.

Stellenbosch University https://scholar.sun.ac.za

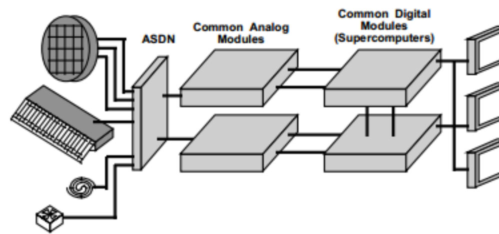


Figure 2.3: Integrated modular avionics [1].

IMA platforms are also increasingly establishing a trend in the aerospace industry to use general purpose computing hardware as the avionics platforms [16] in order to address some of the limitations of custom design [17]. Using this concept, the platform does not perform any dedicated flight control functions but provides the necessary computing, communication and memory resources to implement the required functions. The use of commercial-off-the-shelf (COTS) technology leads to a system that is mirrored on industry standards [18], with the additional benefit that design time and cost are also reduced.

The development of sophisticated mobile handsets has spurred the development of high performance, low-power and low-cost processors that deliver high floating point calculation performance. Such characteristics also make them well suited for use in research-based unmanned flight control system hardware. Various appropriately equipped low-cost, low-power and low-weight rapid prototyping boards are currently available that could be used as general computing hardware for a flight control system. The use of high-performance computing hardware would also make it possible to use a (real-time) OS on top of the hardware, similar to what is being done in IMA systems. An advantage of introducing an OS into the system is that it abstracts away a significant amount of the "bare-metal" firmware programming - resulting in software that is also more portable and reusable, since it is less dependent on the underlying hardware.

2.2 Distributed Systems

It could reasonably be expected that autonomous flight research would focus on a variety of mission types and objectives - each one of which could require the use of a different set of technologies. It could also be expected that improvements in electronics hardware and sensor technologies need to be continuously exploited to achieve research objectives. The combination of these factors indicate that there will be a need for frequent replacements/redesigns of the current avionics, if the existing design approach continues to be pursued

Stellenbosch University https://scholar.sun.ac.za

[19]. By adopting a distributed systems architecture for the avionics system, a flexible substitution and extension of subsystems could be enabled [20] [21].

Tied to the introduction of a distributed systems architecture is the use of a suitable data bus. CAN has to date been the standard data bus in use in avionics systems in the ESL due to its deterministic nature. However its low-speed and low bandwidth could limit the ability of a flight control system to support high bandwidth / throughput applications. Advances in computer networks have established Ethernet as an ubiquitous standard, with COTS parts being readily available. The adoption of Ethernet in the past in real-time, embedded systems as a data bus has been very limited due to the non-deterministic behaviour of its access contention resolution algorithm when being used in conjunction with a shared medium. This problem has been addressed with the development of switched Ethernet. Its use in industrial applications has been validated [22]. A commercial standard describing the use of deterministic Ethernet, ARINC 664 as well as a vendor implementation thereof, known as Avionics Full Duplex Ethernet (AFDX) also exist and are gaining support. It is even possible to develop an AFDX implementation entirely in software on top of an existing switched Ethernet network [23].The use of switched Ethernet as a data bus would be a significant enabler since:

- Higher volumes of data could be transferred at a greater speed between modules connected to the data bus, allowing the use of new "high bandwidth" sensors.
- The flight control system could be extended with dedicated payload modules that provide new capabilities for mission-specific tasks.
- The need for more processing power could be addressed by enabling the capability to add more computing modules to the data bus and spreading the workload across the various modules. This implies the use of a new software programming paradigm that enables this facility.

2.3 Software

Developing software for a high-performance, real-time embedded avionics platform is a complex task, made more difficult if it is based on a distributed, modular architecture. An accepted technique for developing complex software systems is to use the object-oriented (OO) programming paradigm. Object-oriented techniques focus on composing applications from classes and objects that have well-defined interfaces and are related via derivation, aggregation and composition [24]. However, using OO techniques still results in an amount of careful and error-prone work that is required when connecting many small

Stellenbosch University https://scholar.sun.ac.za

parts together in order to form a module handling some bigger part of an overall application [24]. In general, a lot of boilerplate code is still required.

On the other end of the spectrum are automatic code generation suites such as IBM Rhapsody, Esterel Technologies SCADE and MathWorks Simulink / Stateflow combined with the Embedded Coder extension. In the case of these tools, software is developed with graphical building blocks in a model-driven design environment. Once the building blocks have been connected to form a complete system, an automatic code generator is able to turn the model into equivalent C source code. The generator in turn has been verified to generate safety critical code without any ambiguity according to a specification such as DO-178B / DO-178C, allowing the automatically generated software to be used in airborne environments. As can be expected, these tools are available at a price premium.

Research was conducted to identify possible solutions that would deal with the complexity of object-oriented software development for distributed, real-time embedded systems, while at the same time being cost-effective. A notable development evolved from a program started in 1995 at McDonnell Douglas, now part of the Boeing Company, to study the possibility of reusing flight software that was being developed for its various fighter aircraft [25]. The goals of the project were to:

- enable the development of new control applications that were difficult to implement using software techniques available at that time.
- define a system architecture that is based on COTS hardware and open software, standards.
- develop a re-usable software architecture framework.
- develop applications using reusable software components that reside in the framework

The software framework contemplated at the time would ideally contain the change and maximize re-usability of software across their product lines. The project was later expanded under the Open Control Platform initiative in the Boeing Phantom Works division to also include UAVs [26]. The software framework that was evolved out of these and other similar studies came to be known as middleware [2].

Middleware is located between the application being developed and the underlying OS / network / hardware. It provides re-usable building blocks, infrastructure and services that can be used to compose application software -

Stellenbosch University https://scholar.sun.ac.za

as a result, it is more completely referred to as host infrastructure middleware. It allows developers to build applications from a re-usable software framework instead of being concerned with developing low-level code that has a direct dependency on the underlying OS and hardware platform. Examples of well known host infrastructure middleware include Oracle's Java virtual machine and Microsoft's Common Language Runtime - the foundation for .NET services.

A specialized type of middleware, known as distribution middleware, has been developed that extends the functionality of host infrastructure middleware by providing re-usable network programming capabilities and services. Distribution middleware makes it possible to create and program distributed systems that run across computer networks. Examples of distribution middleware include Microsoft's Component Object Model and the Common Object Request Broker Architecture (CORBA) from the Object Management Group (OMG). It is interesting to note that the use of CORBA in real-time control applications for robotics and UAVs has been validated [27] [28].

Both commercial as well as free, open-source software (FOSS) versions of middleware that are suitable for use in a distributed, real-time and embedded environment are available and actively maintained. Commercial versions of distribution middleware are available from companies such as Prismtech and RTI. A FOSS version of distribution middleware, directly developed as a result of the research started at McDonnell Douglas / Boeing in collaboration with Washington University is also available [29] [30]. It is also known that proprietary derivatives of the FOSS distribution middleware exist and are in active use at companies such as Northrop Grumman [31].

Middleware can be thought of as being organized in layers, as shown in figure 2.4. The host infrastructure middleware sits on top-of and depends on the underlying OS and hardware. Distribution middleware extends host infrastructure middleware with distribution / networking capabilities and to this end provides further re-usable services, such as an event service, which will be described further in 3.3.

Stellenbosch University https://scholar.sun.ac.za

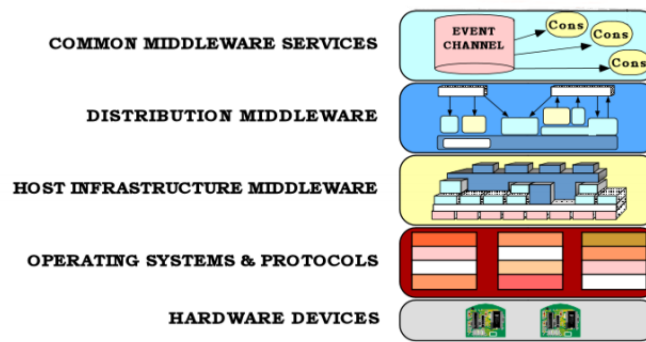


Figure 2.4: Middleware layers [2].

Using a suitable distribution middleware framework would make it possible to create complex application software - in this case flight control software - that can, as much as possible, contain change and enable re-usability while at the same time also allowing the system resources to be expanded [32] [33]. It is claimed that the use of middleware can reduce the effort required to build and maintain distributed, real-time and embbedded (DRE) systems [34].

2.4 Components

Domain independent distribution middleware has been developed that is suitable for use in a distributed, real-time and embbedded (DRE) environment [35]. It is based on the CORBA architecture that has been standardized by the OMG. More recently, the standard was evolved to support component-based system development. In the context of middleware, components are software units that enhance the object oriented programming paradigm by encapsulating parts of a system that are offering a specific service. The service is exposed and made available to other components via attributes and ports. Component-based software development proceeds by assembling various components into a system, using their exposed ports to tie them together.

When used in the context of software development, the term component is often ambiguous [25]. For the purposes of this project, a component will be a software entity that:

- can be composed of multiple smaller objects.
- provides a set of services to a client
- executes inside a server runtime / container which is part of the middleware framework.

Stellenbosch University https://scholar.sun.ac.za

- relies on few underlying assumptions about the environment in which it is executing.
- has access to reusable middleware services such as an event channel, a persistence service, a load balancing service, scheduling mechanisms.

2.5 Standards

A lightweight profile of the component-based distribution middleware, known as Lightweight CORBA Component Model (LwCCM) has been standardized by the OMG [36]. It retains the essential functionality to develop distributed, real-time systems but is optimized to be used in a resource constrained, embedded environment. Domain-specific component models have also been developed: these include research to combine the LwCCM component model with the services described in the ARINC-653 standard in order to build a hard real-time platform suitable for use in safety-critical avionics [37]. Other domain-specific examples include Autosar for the automotive industry and Orocos, which is used in the robotics domain.

The current state of the art is for component-based distribution middleware to be combined with publish/subscribe technologies in order to enable the exchange of large amounts of data efficiently. An example application of this architecture was described in the development of a fractionated satellite system in which a traditional monolithic satellite was replaced by a cluster of wirelessly connected modules [38].

Presently, work is progressing on a new standard, the Unified Component Model (UCM), which will evolve out of the LwCCM. The main goal of the new UCM standard will be to remove the dependency on CORBA and introduce more support for publish/subscribe technologies. It is hoped that this will result in an even simpler, more extensible and customizable middleware.

2.6 Model Driven Development

The use of software components in the development of the flight control system opens up the possibility to define system functionality using a Model Driven Architecture (MDA) methods, the specification of which has been standardized by the OMG [39]. Using an MDA approach, it is possible to design the interface, interconnectivity and configuration of software components in a visual modelling environment such as the free, open-source Generic Modelling Environment (GME) from Vanderbilt University [40]. Once the software model has been completely developed it is possible to generate component interface definition, deployment and configuration files for the system as well as low-level

Stellenbosch University https://scholar.sun.ac.za

networking code that are usually required by middleware to run and interoperate the software components. This task could be tedious and error-prone, since a manual approach would entail editing configuration files by hand.

While tools like Esterel Technologies SCADE and MathWorks Matlab Simulink / Stateflow allow C source code to be developed for the complete system, GME attempts to strike a middle-ground by automating away the most tedious, repeatable tasks. The implementation of the business logic of a flight control system however must still be completed manually.

2.7 Real-time Operating System

The correct operation of the flight control system will require the ability to deliver computational results in a deterministic manner. This cannot simply be achieved by using a particular type of "real-time" middleware, since the middleware relies on underlying OS facilities that enable deterministic performance. In order to achieve real-time performance, a mechanism in the OS must be available that enable highest priority tasks to obtain use of the CPU without being pre-empted by any other tasks, including the OS kernel itself. On such an OS, known as a real-time OS (RTOS) the latency of execution of a task can only be affected by tasks with a higher priority.

At least two different distinctions of real-time behaviour exist: hard and soft real-time. Hard real-time guarantees are important for applications such as flight control systems: these systems require that every task that is executed must meet a deadline for execution, or a critical failure could occur. In contrast, it is acceptable for some deadlines to be missed in a system requiring soft real-time deadlines. However, eventually performance could degrade if too many deadlines are missed.

The overwhelming majority of RTOS's that are available for avionics system are of the commercial variety - examples of which include LynuxWorks LynxOS-178 and Wind River's VxWorks 653. As their name appears to suggest, they have been certified to be suitable for use in airborne systems requiring DO-178B/C certification and ARINC-653 compliant partitioning mechanisms that isolate running processes from each other. Unfortunately, RTOS's providing these certifications and facilities are only available at a price premium.

A trade-off can be achieved with the Linux kernel by forgoing the certification status and partitioning mechanisms that are provided in commercial RTOS's and only requiring that the OS meet hard real-time schedules. Unfortunately, a standard Linux kernel will not be sufficient as it only meets soft real-time requirements - the latency of execution of a task could result in a

Stellenbosch University https://scholar.sun.ac.za

deadline being missed. This could result in catastrophic failure of the flight control system. However a kernel patch, known as the `PREEMPT_RT` patch [41], converts Linux into a fully preemptible kernel, thereby conferring hard real-time capabilities on the OS. A `PREEMPT_RT` patched Linux kernel has been shown suitable for use in hard real-time control applications [42].

Developing a real-time capable system that is at the same time also based on a distributed architecture could be a difficult task if facilities to enable these qualities had to be developed from scratch. As will be seen in chapter 3, the distribution middleware provides re-usable infrastructure and services, including mechanisms to enable real-time scheduling of tasks, that enable a DRE system to be built.

2.8 Compatibility

In order to retain support for existing sensor and actuator technologies that have been developed, a hardware gateway can be introduced that translates between the CAN and Ethernet buses. This mirrors a trend in industry for to be translated from/ to "legacy" transducer equipment into the new standard data bus network [16]. A difficulty with this approach could be to coordinate the operation of the many different sensing and actuating devices [2].

2.9 Conclusion

A high level overview of the approach that will be taken to develop a new flight control system architecture has been provided. Implementing a distributed, real-time and embedded flight control system that addresses the problems of the existing systems in the ESL can be achieved by combining existing solutions:

- A networked system will provide the ability to address the existing computation and communication resource constraints.
- Standards based distribution middleware such as the Lightweight CORBA Component Model will provide the software infrastructure and programming paradigm required to develop the flight control algorithms on a resource constrained, distributed system [2].
- Real-time capability will be enabled on general purpose COTS computing hardware by patching a Linux OS with the RT-Preempt patch.
- Compatibility with existing transducer hardware will be maintained by introducing gateway devices that translate between CAN and Ethernet buses.

Stellenbosch University https://scholar.sun.ac.za

This chapter outlined modern approaches, based on open industry standards, that can be used to design a replacement flight control system suitable for unmanned flight research that will address the limitations of the existing systems. In chapter 3 these approaches will be developed further into an architecture for the new flight control system.

Stellenbosch University https://scholar.sun.ac.za

Chapter 3

Architecture

In chapter 2 the main concepts were introduced which will aid in the design a flight control system to meet the project objectives. In this chapter, the concepts are expanded in order to develop an architecture for the avionics system.

3.1 Structure

The architecture of a flight control system can be divided into two concerns: how the system maintains a model of the state of the aircraft and the manner in which updates in response to new input data are propagated through the system [3]. To illustrate this better, consider the operation of a flight control system, which can be briefly and generically described as continuously performing the following functions:

- collect input data, including data about the position and environment, using sensors.
- use the input data to estimate the actual state of the aircraft.
- compute the desired aircraft state with respect to a guidance mode, such as autopilot.
- manipulate aircraft actuators to bring the actual and the desired states closer together.

When viewed from a structural perspective, this description of a flight control system can be interpreted as an example of a Model-View-Controller (MVC) architecture [25] [43]. The system maintains a model of the actual state of the aircraft that is controlled via updates in response to new input data. The updates are "viewed" by the aircraft effectors, which bring the aircraft in

Stellenbosch University https://scholar.sun.ac.za

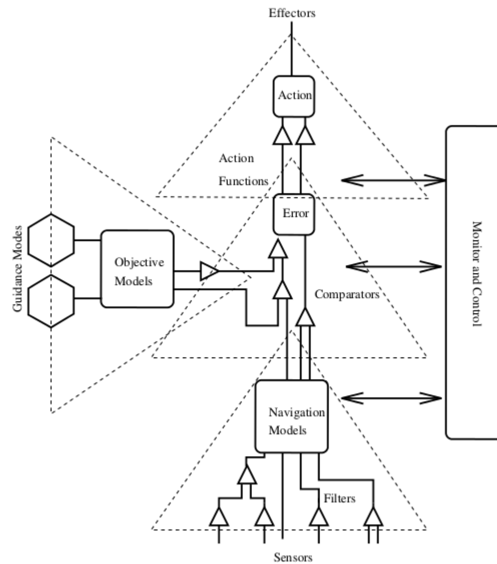


Figure 3.1: Template Flight Control System Architecture [3].

closer agreement with the desired state. Model maintenance, input sources and output sinks are generally handled by different components in order to keep their responsibilities separate.

The diagram shown in figure 3.1 summarizes the idea of an avionics system based on a MVC architecture: sensors feed new input data into the system via filters, which may establish average values or remove any gain/bias that was introduced in the measured value during sampling. Navigation models represent the actual state of the aircraft, as updated by new sensor values. Objective models represent the desired state of an aircraft, which are informed by guidance modes such as autopilot, way-point navigation or terrain following modes. Helper components may also be introduced when required for estimation purposes [3]. The difference/error between the actual and desired state ultimately drives the the aircraft effectors to bring these two states closer together. Finally, the monitoring and control component allows the system to collect telemetry data and/or activate de-activate the system from a ground station.

The above description alludes to several software entities and mechanisms in the structure of the flight control system. Components represent software entities that contain the necessary core functionality and services that will be used to implement the system. Less obvious, a mechanism to move execution from one component to another is also required. Finally, input data, which is required to calculate updates, also needs to be propagated through the system in an effective manner.

Stellenbosch University https://scholar.sun.ac.za

3.2 Components

As introduced in section 2.4, a component is a software entity which can be constructed from multiple smaller objects in order to build a specific set of functionalities required in a system. It is described by a standard: for the ESL avionics that have been ported and re-implemented in this project, the LwCCM standard was used [36], due its suitability for use in a DRE environment and domain independent applicability.

Components are contained within the middleware framework and run inside a server runtime, also known as a container, that provides it with access to all available middleware services - services such as scheduling, event channels (see 3.3) and others. A component container is similar in concept to a Java virtual machine, in that it enables the component to execute within a well-defined environment, using well-defined interfaces. The container itself is responsible for mediating interaction with the underlying OS. In this way, it is possible to develop a single component that can be used on a range of operating systems.

The functionality contained within a component can be shared with other components. For this purpose, the LwCCM standard enables components to expose available operations / services and functionality through port and attribute interfaces that can be specified when the component is created. Ports allow components to interact and collaborate with other components, by sharing operations that it is able to perform. Alternatively, operations can be requested from other components through ports. The types of ports available are shown in figure 3.2. Facets are ports through which functionality contained within a component can be accessed. A receptacle enables a component to express the need for functionality that it does not implement, but requires in order to correctly perform its own function. Finally, attributes represent values configured within a component that can be retrieved or altered by other components.

Stellenbosch University https://scholar.sun.ac.za

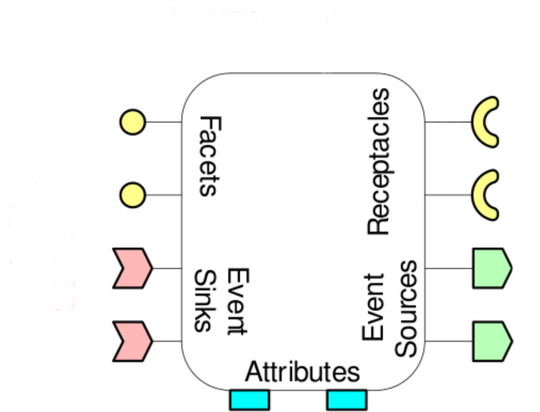


Figure 3.2: Ports and attributes of a software component

Event sinks and sources are ports through which event notifications can be received and sent. Events are signals/messages which a component can use to notify other components of an event that has occurred. They enable a publish/subscribe ability in components, in which publishers of a message are not aware of any subscribers to the message. Subscribers on the other hand can express interest in a message by subscribing to receive a particular type of message. The use of the publish/subscribe mechanism promotes looser coupling between components and also enables to be distributed more dynamically across a network topology.

Component interfaces are described in a programming language-independent manner using the Interface Definition Language (IDL). The IDL-defined interface represents the ports and attributes of a component that should be advertised and made available to other components. This allows the interface and implementation of a particular object to be separated. Once the interface of a component has been specified in IDL, a compiler which forms part of middleware generates a programming-language specific skeleton implementation of the component known as the executor. The skeleton implementation / executor can then be filled out with application logic. Figure 3.3 shows how an example minimal IDL to C++ mapping would look like. Besides the executor, the IDL compiler also generates another software entity known as the servant. The servant contains all the methods required for components to interact in a distributed system, including methods to handle remote method invocations. The servant code rarely has to be modified by the developer.

Stellenbosch University https://scholar.sun.ac.za

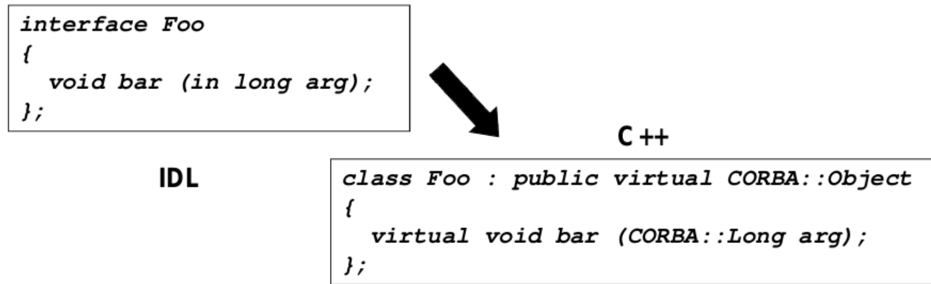


Figure 3.3: IDL to C++ mapping

The IDL for a minimal rate generator component, shown in the three component flight control system of figure 3.4, could be implemented as shown in the IDL listing below:

```
eventtype tick
{};
component RateGenerator
{
    publishes tick Pulse;
    attribute long Rate;
}
```

Once all components have been programmed with suitable business logic implementing their functionality, they can be combined to form a system by connecting their exposed interface ports. Facets (offered methods) and event sources of a particular component can be connected to receptacles (required methods) and event sinks of another component. Figure 3.4, which represents a very basic flight control system, illustrates this idea.

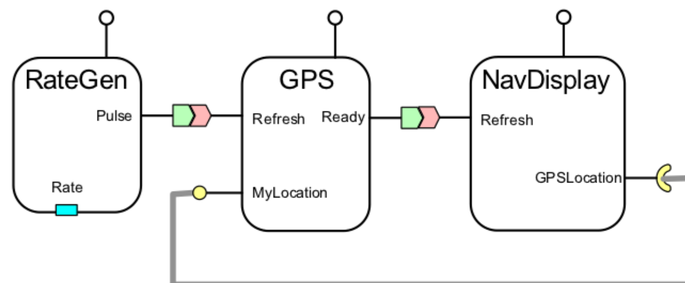


Figure 3.4: A basic flight control system

Stellenbosch University https://scholar.sun.ac.za

The `RateGen` component pulses the `GPS` component at a rate configured through the `Rate` attribute. The `Pulse` event source is connected to the `Refresh` event sink of the `GPS` component, which forwards the event onto the `NavDisplay`. Upon triggering the `Refresh` event, the `GPSLocation` receptacle updates the `NavDisplay` location by retrieving location data from the `MyLocation` facet of the `GPS` component.

3.3 Event Service

The event service implemented in the LwCCM standard - which is a real-time capable service [44] - was employed to enable a publish/subscribe mechanism that decouples publishers which periodically generate data and notifications from subscribers that consume the data. The event service is a re-usable facility that is offered by the distribution middleware.

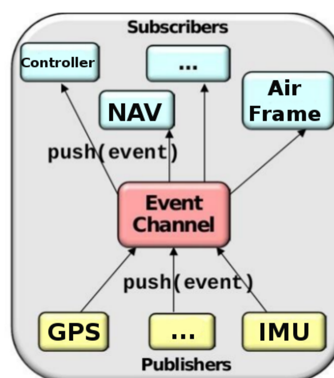


Figure 3.5: Publish/Subscribe architecture using an event service.

The service enables the software components to send event notifications between each other that alert a interested component of a processing tasks that has completed, or new sensor data that has become available. The use of an event service makes the system easier to evolve, since publishers and subscribers are decoupled from each other. It also makes the system easier to schedule, as will be described in 3.4.2.

3.4 Flow Policies

It was found that the propagation of updates through an avionics system, also known as the flow policy, significantly impact performance, distributability, re-usability and maintainability of a system [45], since they determine how the software communicates and executes. The flow policy of a system can be

Stellenbosch University https://scholar.sun.ac.za

categorized into two concerns: the flow of control, and the flow of data through the system.

Control flow represents the movement of execution in the software system. It can be performed in three different ways[45]:

- synchronously on reception of new input - this mode is used when execution is required as soon as new input data is available.
- synchronously according to system output requirements - this mode is used when execution is required at a periodic rate.
- asynchronously - this mode is used when other timing/periodic responses are required.

Control flow determines when it is acceptable for a particular software component in the system to execute. However, it may be possible that at any point in time more than one software component is able to execute. In order to decide the order in which components will execute, a scheduling policy is used. These two aspects of control flow are also known as component control flow and application control flow respectively [46] [45].

3.4.1 Component Control Flow

Within component control flow, there are two axes of variation that need to be considered when deciding on the avionics system flow policy: the control flow can be either push or pull based and the component itself could either be active or passive.

3.4.1.1 Push or Pull Control flow

In push mode, the component waits for an external actor to effect its execution. The decision when to execute is removed from the component and placed in a separate entity. The advantages of push control flow are:

- avoids polling overhead - an external actor is responsible to initiate execution.
- minimizes latency between execution of components.
- minimizes scheduling logic within components.

In pull mode, the component interrogates the current system state in order to determine whether it should run. One way it could do this is to poll suppliers

Stellenbosch University https://scholar.sun.ac.za

for new data availability and execute upon reception of a positive notification. In this mode, the component itself controls when it executes. Pull control flow results in a component that has a more intelligent / mode dependent execution mechanism - however, the ability to interrogate system state could also result in the component being less reusable. Considering the merits of both push and pull control flow, it was found that push control flow is preferred for an avionics system [45].

3.4.1.2 Active / Passive Components

The other axis of variation that should be considered is whether to implement active or passive components. An active component has its own processing resources, while a passive component relies on other active components to invoke its execution. The advantage of an active component is that it will potentially be more responsive due to being equipped with its own dedicated processing resources. On the other hand a passive component:

- requires less context switching, which is useful for systems equipped with a single processing unit.
- exhibits less resource contention.
- makes the use of smaller components more feasible.
- supports centralized scheduling.

For a DRE avionics system, the advantages of a passive component make it the preferred choice, since these address the objectives of the system closer than those of an active component [45].

3.4.1.3 Component Control Flow Strategy

Combining the the use of push mode with passive components results in the avionics system having an aggregated passive-push approach to moving execution through the system from one component to another. Concretely, this will be achieved by equipping each component with an execute event sink. As soon as an event is received on this sink, component execution will be invoked.

3.4.2 Application Control Flow

Component control flow determines the mechanism by which execution passes from one component to another. As explained in section 3.4.1, the components in the avionics system will implement a passive-push component control flow policy. However, at least one active component is required for the system to

Stellenbosch University https://scholar.sun.ac.za

function, otherwise no possibility exists for the execution of any component to be invoked. Scheduling execution of each component in the system, also known as application control flow, is the second aspect of control flow that needs to be considered.

Since the the new avionics system architecture needed to be ported from, and would be compared against an existing system for correctness, a scheduling design was chosen that results in component execution corresponding to the scheduling of the existing ESL avionics system.

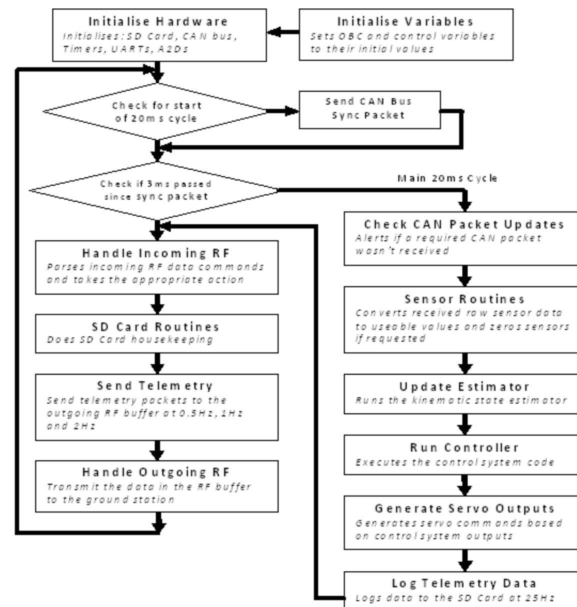


Figure 3.6: Flow Chart of periodic tasks [4].

Figure 3.6 is a flow chart showing the control flow of the main tasks in the existing ESL avionics. As can be seen, the cycle results in servo output being generated every $20ms$. Figure 3.7 provides more detail on the the flow of control and data in an existing flight control system.

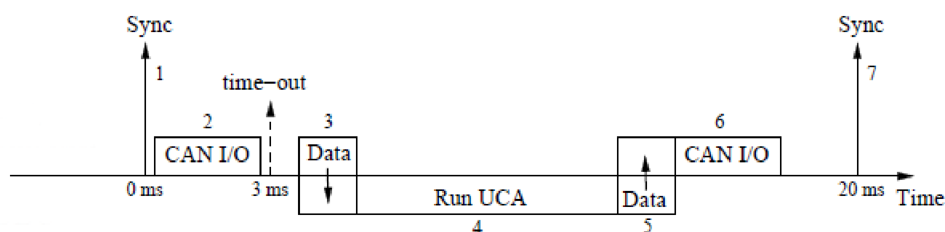


Figure 3.7: Control and data flow diagram.

Stellenbosch University https://scholar.sun.ac.za

A sync is generated every $20ms$ in order to mark the start of a new output cycle. During the first $3ms$ of the cycle sensors sample the environment in order to generate new input data. After the $3ms$ "sensor cutoff" time-out occurs, the avionics system retrieves the new data and runs estimation and control algorithms. The algorithms calculate updated actuator values necessary to retain flight control. The updated actuator values are written out and the cycle repeats.

To mimic this behaviour, a rate generator component will generate sync pulses every $20ms$ and sensor time-out pulses $3ms$ after the $20ms$ cycle has started. The sync pulse will generate execute events that are sent via the event service to the sensor components. Upon reception, the sensor components will generate new data from updated readings. After the $3ms$ sensor time-out occurs, execute events will again be sent via the event service, this time to estimator and controller components, in order to calculate updated state and actuator values. The controller invokes the execution of the servo component that will write out the updated actuator positions, again by sending an execute event through the event service.

Real-time operation of the system on a Linux OS modified with the `PREEMPT_RT` patch is guaranteed by assigning the highest possible absolute priority to the process running the rate generator component and choosing `SCHED_FIFO` as a scheduling policy for the process. As a result, the rate generator component can only be pre-empted by processes that have a higher absolute priority. `SCHED_FIFO` implements a first-in first-out, fixed-priority, real-time scheduling policy as specified in the POSIX standard. `SCHED_FIFO` requires that the active process explicitly yield the processor to allow other tasks to run as well.

Within the avionics system, it is possible to change the scheduling of execution of the various components by controlling the order in which execute events are delivered from the event service to the receiving component [46]. Since all execute events pass through the event service, the scheduling can be updated "behind the scenes", by assigning different priorities to the various dispatched events. Even thought this facility was not used in the project, the distribution middleware provides re-usable schedulers that allow this to be achieved [47].

3.4.3 Data Flow

The flow of data in a flight control system is generally from sensors to effectors. Even "feedback" is obtained from sensors, and not assumed to be the last value that was written out to an effector [3]. Similarly to component control flow, the movement of data through the system that is required for calculation and eventual output can either be push or pull based: in push mode, the publishing component transfers data to a subscribing component. In pull mode, the

Stellenbosch University https://scholar.sun.ac.za

receiving component is responsible for retrieving data from the generating component. Push mode implies that receiving components rely on external actors to satisfy data needs, while in pull mode the receiving components themselves control their data flow. The advantages of push data flow are that:

- latency between component execution start and data reception is minimized when data can be combined with push control flow notification.
- no concurrency mechanisms are required when data is being transferred to a receiving component.

On the other hand, pull mode data flow has the following advantages:

- Receiving components retrieve only the data they need, when they need it.
- Receiving components can potentially control when calculations are performed, since a function call to retrieve data is made to generating components who could use this as a trigger to perform updated calculations.

For a DRE avionics system, it was found that the advantages of pull data mode outweigh those of push data mode [45].

3.5 Distributed Systems

To enable developers to build a distributed system with multiple processor boards, software infrastructure enabling this functionality would be desirable, in order to avoid the difficulties associated with network programming [2]. A software entity that forms the basis of CORBA distribution middleware and is capable of providing this functionality is known as an Object Request Broker (ORB). It is an object oriented equivalent of a previous technology known as Remote Procedure Call (RPC), which allowed a client program to invoke an operation on a program that could potentially be separated via a network. An ORB hides the network programming complexities and protocol-specific details that enable this ability.

An ORB facilitates the routing of the request to perform an operation through the network from client to the object implementing the operation, and returns any result back to the client, as shown in figure 3.8. In a manner, it brokered the interaction between client and object.

Stellenbosch University https://scholar.sun.ac.za

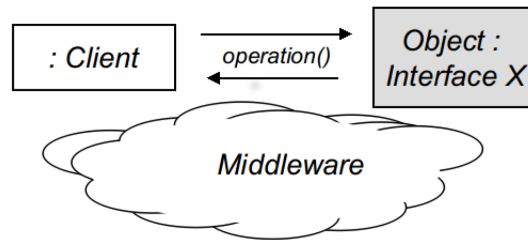


Figure 3.8: Remote method invocation.

An ORB keeps track of the various ports (offered methods/operations) that a component has made available through the interface that was declared in IDL which was used to create the component. The repository of the various operations is then used as a part of the resolution mechanism to determine where a particular request for an operation needs to be routed to.

A mechanism is required for the client to locate the component on which it would like to invoke an operation on - it requires the "contact details" of the potentially remote component. Distribution middleware has a few services that can be used to achieve this, one of them being the interoperable object reference (IOR). Upon instantiation, a component can be configured to write out its IOR into a file. An ORB can then examine the IOR, discover the location and network protocol required to contact the component, and route the client's request appropriately. This process is transparent to the client - nothing in the IOR can reveal the location of the component to the client.

Various implementations of CORBA distribution middleware are available, however not all of them may be suitable for use in a real-time control application. Boeing (McDonnel Douglas), together with Washington University, and funded by DARPA, developed an ORB that was optimized to support distributed, real-time and embedded flight control applications for their fighter jets. The development was later extended to include the needs of their UAV program (Phantom Works) [48]. The ORB that was developed came to be known as "The ACE ORB" (TAO) [29]. TAO was later extended to include support for the component based software development paradigm. The TAO extension providing component support is known as the Component Integrated ACE ORB (CIAO). Together, the open-source TAO and CIAO implementations are compliant with the OMG LwCCM standard, first introduced in section 3.2.

Stellenbosch University https://scholar.sun.ac.za

3.6 Hardware Block Diagram

Section 2.2 described the benefits of building a flight control system in a distributed manner around on an Ethernet network. This capability in the architecture would allow additional computing nodes to be added, thereby increasing the capacity of the system. It would also allow new functionality to be introduced with less disruption to the existing system than a complete redesign, since the new functionality could be located on a separate node that is attached to the network.

The avionics computing hardware will be based on a COTS single board computer (SBC). An important requirement is that it must be able to run Linux OS. Most of the existing actuator and power management hardware would be reused for this project. Since the hardware was built around a CAN bus, a interface board will be developed that translates between the CAN and Ethernet protocols. Alternatively, COTS CAN-Ethernet gateways are also available, although at a price-premium. COTS RS232-Ethernet converters are also available that can be used to translate GPS sensor input toward Ethernet.

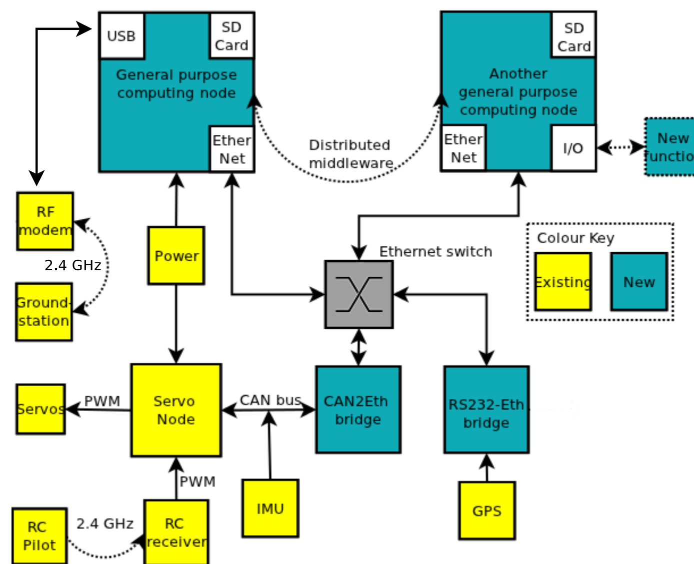


Figure 3.9: Hardware block diagram of the flight control system.

The block diagram shown in figure 3.9 illustrates the layout of the flight control system with the new hardware elements inserted. Blocks shaded in yellow indicate existing functionality, while blue blocks identify new functionality.

Stellenbosch University https://scholar.sun.ac.za

3.7 Software Block Diagram

Figure 3.10 illustrates a component diagram of the flight control system that has been developed using the MDA approach described in 2.6.

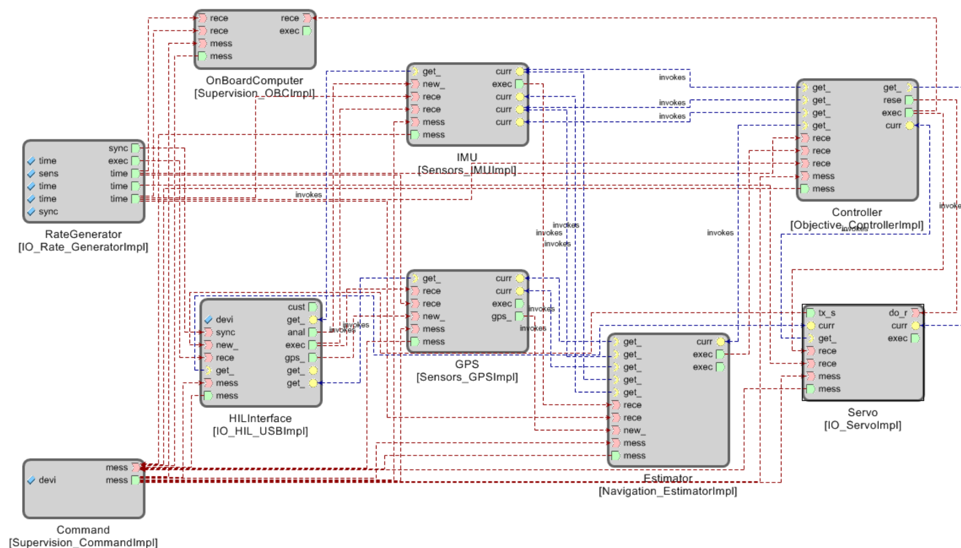


Figure 3.10: Component diagram of the flight control system.

As can be seen, the avionics system consists of 9 components:

- `RateGenerator`: a component implementing scheduling.
- `Command`: a component that provides communication to and from the groundstation.
- `OnBoardComputer`: a component that collects and sends telemetry relating to the functioning of the system.
- `HILInterface`: a component that receives sensor values from, and sends actuator values to a HIL simulation.
- `IMU`: a component representing inertial measurement unit functionality containing pressure-meter, accelerometer, gyroscope and magnetometer.
- `GPS`: a component representing measurements derived from a GPS sensor.
- `Estimator`: a component that uses sensor values to estimate the current state of the aircraft.

Stellenbosch University https://scholar.sun.ac.za

- `Controller`: a component implementing the control algorithms to maintain autonomous flight.

- `Servo`: a component that converts the actuator positions that have been calculated by the `Controller` into suitable timer values that can be sent to the actuators.

The system contains two active components: a rate generator component responsible for driving the real-time scheduling of the system, as described in section 3.4.2, and a command component that represents the interface of the flight control system with the ground-station. The command component has a `message_to_component` / `message_from_component` event source / sink in order to facilitate transfer of messages between the groundstation and components. To interface with the rate generator and command component, each passive component has some common functionality:

- a `receive_execute` event sink to allow execution of the component to be invoked.

- `message_from_command` and `message_to_command` event sinks and sources to enable communication with the command component.

Additionally, the passive components each have one or more `receive_timeout` event sinks that enable it to receive time-out events from the rate generator. The purpose of these is to notify components to send telemetry data to the command component. The remaining ports are dedicated to the specific functionality of the component and will be discussed further in 4.

3.7.1 Configuration and Deployment

Since the flight control system will contain a number of components, it will be necessary to specify the components attributes as well as the interconnections between the various ports that will form an entire running system. Should more than one processor board be used, it will also be necessary to specify on which host the component will reside. To handle these aspects, the OMG has developed a Configuration and Deployment specification [49] which has been implemented in the distribution middleware used for this project [50].

Configuration of the system is achieved by specifying running component instances, their implementation artefacts, as well as interconnection and attributes in an extensible mark-up language (XML) file. Deployment in the context of a component-based system is the task of getting the complete system up and running. Since these task could become tedious with the possibility

Stellenbosch University https://scholar.sun.ac.za

of hard-to-trace errors occurring, MDA tools allow for the required configuration and deployment files to be generated from suitably completed visual models. Suitable tools provided by the distribution middleware then read in the XML configuration and deployment file and appropriately configure and instantiate the various components.

3.8 Simulation

HIL simulation provides the ability to verify that developed functionality is performing as expected, by testing the flight control system against a known mathematically correct simulation of the entire dynamic system. This is achieved by inserting the hardware containing the flight control system into a simulation loop and feeding it with emulated sensor values. The flight control system calculates updated actuator positions based on the provided sensor input and feeds these back into the simulation. Using the actuator values, the simulation is updated by a time increment corresponding to the output rate of the system - in the case of the ESL avionics this is $20ms$. During this time the vehicle dynamics are propagated forward in time by $20ms$ and new sensor values are calculated. The entire cycle is then repeated.

The HIL simulation setup for the existing flight control systems required the use of a distribution board [10], as shown in figure 3.11.

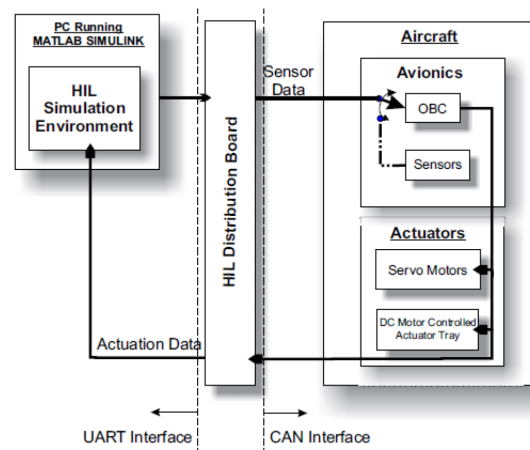


Figure 3.11: HIL simulation of existing FCS [11].

The simulation is implemented in MATLAB Simulink and executes on a PC. Since the sensor values will be distributed via Ethernet to the computing node running the flight control algorithms in the new architecture, it is now possible to feed the computing node with HIL emulated sensor values directly,

Stellenbosch University https://scholar.sun.ac.za

by redirecting the Simulink I/O to occur via Ethernet instead of the serial interface. Consequently, the need for a HIL distribution board was eliminated.

The User Datagram Protocol (UDP) was used to implement Ethernet-based I/O, since it incurs less delay when sending data compared to TCP and also requires less processing at the transmitting and receiving hosts. UDP does not guarantee delivery of packets, but this is less of a concern in a closed system such as the one being designed for the avionics.

3.9 CAN-Ethernet Gateway

In order to support existing sensor and actuator hardware, while at the same time also moving toward a distributed architecture, it was necessary to introduce functionality that would translate between the existing Controller Area Network and the new Ethernet based network. The CAN-Ethernet gateway will fulfil this function and is developed in detail in section 4.1.3. Though several examples of commercial products are available, their price was quite prohibitive for research purposes and it was decided to implement this functionality on a small, lightweight prototyping board that contains both CAN and Ethernet interfaces.

3.10 Conclusion

A hardware and software structure was developed in this chapter for a flight control system that is capable of meeting the objectives listed in section 1.3. The functions that form part of the architecture have been described. They will be developed in detail in chapter 4. Combining all the concepts introduced so far, the avionics architecture can be summarized as follows:

- A distributed architecture based around a switched Ethernet network will be used.
- Distribution middleware will provide the software infrastructure required to develop the flight control system.
- Components will be used to contain the core functionality of the system.
- Component interfaces are described in IDL.
- An event service will be used to aid with the control flow of the system and to distribute events between publisher to subscribers.
- Components will be passive - they will not contain their own processing resources.

Stellenbosch University https://scholar.sun.ac.za

- Execution will be moved from one component to another by sending an execute event to the component.
- Scheduling will be performed by an active, rate generator component. Its process has a high priority and uses the `SCHED_FIFO` scheduling policy, in order to avoid being pre-empted.
- Components will pull data required for calculations on demand.
- ORB to handle "low-level network details and enable distributed systems development.
- A CAN-Ethernet gateway to allow existing transducer equipment to be used.
- The avionics hardware will make use general purpose COTS SBC boards.
- A `PREEMPT_RT` patched Linux OS will be used to provide real-time capabilities.
- For HIL simulations, the existing HIL distribution board will no longer be required.

Stellenbosch University https://scholar.sun.ac.za

Chapter 4

Development

Chapter 3 described the structure and operation of the avionics system architecture. In this chapter, the individual hardware and software functions of the architecture will be developed. The avionics hardware will be chosen and the CAN-Ethernet gateway, required to support existing actuator hardware, will be developed. On the software side, the individual components that constitute the system, as described in section 3.7, will be developed.

4.1 Hardware

4.1.1 Avionics

As shown in section 3.6, the avionics system computing hardware will be based on a COTS SBC since numerous specimens have recently become easily available. The board should have Ethernet connectivity in order to plug it into the system network. It should also have an option to connect serial devices, since the communication protocol between ground-station and the flight control system will be re-used from existing ESL systems (reproduced for convenience in A.1). The SBC must have driver support for all its peripherals and be powerful enough to run a recent version of a Linux-kernel based OS.

Computer-on-Module (COM) based SBCs are a promising hardware platform since the I/O peripherals and connectors are usually located on a separate carrier board to the main computer system. The processor system is mounted on top of a carrier board implementing the required I/O interfaces. This enables an additional upgrade path, since the carrier board can be kept constant while the processor board is upgraded. A commercially available example of a COM system, the Gumstix Overo platform, was selected for this project since its weight ($35g$), form-factor ($105mm$ x $40mm$) and power requirements ($250mA$ at $4v$) lend it to be used in small UAV applications and also compare favourably to the characteristics of the existing system. The processor

Stellenbosch University https://scholar.sun.ac.za

board was equipped with an ARM Cortex A8 core running at 800MHz and 512 Mbyte of random access memory (RAM).

4.1.2 Ethernet Switch

A TrendNet TE100-S5 5-port COTS 100 Mbps Ethernet switch, of the readily available variety, was chosen. Besides being small and offering standard features, the switch did not possess any other notable properties. The switch required a power source capable of supplying 5v at 1A.

4.1.3 CAN-Ethernet Gateway

The LM3S8962 board from Texas Instruments was used to prototype the CAN-Ethernet functionality, since it contained both CAN and Ethernet interfaces available in a relatively small form factor (11.5cm x 6.2cm). The gateway was designed to implement a state machine that translates CAN packets into UDP packets and vice-versa. UDP was chosen since, as with the HIL simulation, it incurs a smaller sending delay compared to TCP. For the gateway to be as efficient as possible, the smaller processing overhead associated with UDP was also desirable. Introducing the CAN-Ethernet gateway into the system would introduce a new delay into the system, but this would be minimized by using UDP.

The CAN-Ethernet gateway was prototyped on an Advanced RISC Machines (ARM) Cortex-M processor equipped evaluation board containing CAN and Ethernet interfaces. On power-up, the state machine cycles through a few states that initialize the Ethernet interface, interrupts and the Internet Protocol (IP) stack, resulting in the board acquiring an IP address via DHCP. If no DHCP servers can be found, the Auto-IP module is used, which enables a server-less method of assigning an IP address. At this point, the gateway sends out UDP broadcast messages to find targets interested in receiving its UDP packets encapsulating CAN data. These could either be another CAN-Ethernet gateway, in which case the UDP packet is unpacked and sent out on the receiving gateway CAN interface (CAN-over-IP functionality) or it could be a flight control system interested in receiving and processing the CAN data.

The CAN-Ethernet gateway enters a state waiting for CAN or UDP messages to arrive, once targets have expressed their interest in receiving UDP messages from the gateway by replying to the broadcast message. Incoming CAN messages trigger an interrupt service routine which copies the CAN data into a ring buffer. An exception that is unique to the Cortex-M, known as a pended software interrupt (`PendSV`), is then raised, which notifies the system that new CAN data is available that can be sent out via UDP. `PendSV` is an asynchronous exception that is only triggered once all higher priority interrupt

Stellenbosch University https://scholar.sun.ac.za

service routines have completed processing. In order to ensure that incoming CAN messages and Ethernet messages can always be timeously processed, the CAN and Ethernet handling service routines have been assigned with highest priorities, while `PendSV` has been assigned the lowest priority.

The `PendSV` service routine removes any available CAN messages from the ring buffer and packages them into a UDP packet containing the CAN message ID and data, along with flags indicating whether the CAN message originated from an extended frame and/or remote transmission. The UDP packet is prepended with a unique string identifying it as a gateway data packet and sent to all targets that responded to the initial broadcast messages.

Incoming UDP messages that contain CAN data are unpacked immediately. A CAN message is then crafted from the contents of the UDP message and sent out on the CAN interface.

4.2 Real-time Operating System

Though several Linux distributions are freely available that include the necessary firmware and device drivers to be suitable for use on a Gumstix platform, none of them included a recent `PREEMPT_RT` patched Linux kernel, even though `PREEMPT_RT` patches are produced for every Linux kernel version. To retroactively patch the kernel could also be a daunting task, since it was not known which options were used when first compiling the kernel.

To make the process of obtaining a Real-time Linux Operating System repeatable and modifiable, in case a new processor board was introduced, a set of tools from the Yocto project were used [51]. The Yocto tools enable the creation of a Linux-kernel based OS that is customized according to the requirements of the underlying hardware. This is achieved by specifying a set of recipes that are "compiled" to eventually build the desired OS image. Recipes to include the correct board support packages and Linux kernel patches can also be added, thereby automating the task of customizing the OS for the targeted hardware platform. The Yocto organization creates packages for most commercially available hardware platforms. Once the OS has been "compiled" it can be transferred to suitable bootable media that the hardware platform accepts.

Stellenbosch University https://scholar.sun.ac.za

4.3 Software

4.3.1 Method

Figure 3.10 provided an overview of all the components that form part of the flight control system software. As shown in section 3.7, the system consists of 9 components: `RateGenerator`, `Command`, `OnBoardComputer`, `HILInterface`, `IMU`, `GPS`, `Estimator`, `Controller` and `Servo`. The process of developing a software component starts by declaring the component's interface using the programming language agnostic IDL specification. The IDL files are compiled to generate a skeleton implementation of the component that can be completed by the developer with the required functionality (described in section 3.2). In this section, the function and structure of each of the components will be developed in turn by describing the interfaces, the function of the individual ports in the interface and how they are used to interact with other components. The complete IDL interface descriptions of all components can be found in an archive attached to this report.

A modification to the Matlabl HIL simulation software was also required, since the avionics system would now be Ethernet based. As a result, emulated sensor values could no longer be transferred via the previously used, USB-based HIL distribution board, but had to be transferred directly from the simulation to the processor board responsible for sampling sensor values.

4.3.2 RateGenerator

The `RateGenerator` is an active component that is responsible for performing the application control flow of the flight control system. Real-time performance characteristics are obtained by setting the highest possible priority of its containing thread and using a `SCHED_FIFO` scheduling policy, as described in section 3.4.2.

The rate generator, as shown in figure 4.1, contains five event sources () in order to help it schedule events.

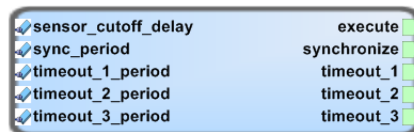


Figure 4.1: Rate Generator component interface.

The attribute () values can be used to configure the timing of the triggered events. In total, five different events are configured using the attribute

Stellenbosch University https://scholar.sun.ac.za

values:

- The `sync_period` attribute configures the rate at which sensors are sampled and output is provided to the actuators. The attribute value was set to $20ms$, which configured a `sync` event to be sent every $50Hz$.
- The `sensor_cutoff_delay` attribute determines how long after the start of a sync period the sensor cut-off event is triggered, at which point the sensors stop sampling. The attribute value was set to $3ms$.
- The `timeout_1_period` attribute controls the signalling of `Controller`, `GPS` and `OnBoardComputer` components to send telemetry data via the `Command` component to the ground station. The attribute value was set to $2s$.
- The `timeout_2_period` attribute results in an event being pushed to the `Servo` component to compel it to send telemetry data. The attribute value was set to $1s$.
- The `timeout_3_period` event signals to the `Estimator` and `IMU` to send their respective telemetry values. Additionally, this event also pushes the `Controller` and `OnBoardComputer` to send their secondary telemetry data.

Internally, the `RateGenerator` uses a reactor object [52] to schedule five timers [53] corresponding to the previously described attributes. Upon time-out of each, the reactor calls the associated handler method, resulting in the corresponding event being pushed: at time-out of every `sync_period`, the `sync` event indicates to sensors to start sampling the environment in order to generate new input data. Actuator positions that were calculated in the previous cycle are written out. Shortly thereafter, the `sensor_cutoff_delay` time-out is triggered which results in the `execute` event being sent that invokes execution of the passive components. The components use the new sensor input data to perform their respective functions. `GPS` and `IMU` components are executed first, followed by the `Estimator`, `Controller` and `Servo` components. Finally, telemetry data is sent upon reception of events associated with the previously described attributes.

4.3.3 Command

The `Command` component is an active component that is the interface of the flight control system to a ground-station. The function of the `Command` component is to relay messages such as telemetry data from the components to the ground-station. It also receives data from the ground-station and distributes

Stellenbosch University https://scholar.sun.ac.za

it toward the components. For this reason, it is an active component since it needs to be ready to receive and transmit messages to and from the ground-station at any moment. The interface of the `Command` component is shown in figure 4.2.

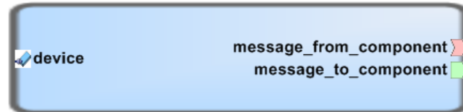


Figure 4.2: Command component interface.

The `message_to_component` event source (▢) enables the `Command` component to send event notifications, including any ground-station data that may have accompanied the notification, toward the passive components. Passive components in turn only act on notifications that are relevant to them.

The `message_from_component` event sink (▢) enables passive components to send notifications, such as telemetry data, back to the ground-station via the `Command` component.

Internally, the `Command` component uses RS-232 serial communication toward the ground-station. During flight, the link will be maintained by a RF modem. When a HIL simulation is performed, the wireless link can be replaced by a serial cable between the ground-station and the avionics system . The `device` attribute (▢) can be used to configure the serial device that will be used. Its value is set to `/dev/ttyO2` by default. This value maps directly to the underlying Linux OS serial device.

4.3.4 HILInterface

The `HILInterface` is the component that interfaces with a Matlab Simulink simulation in order to receive emulated sensor values from the simulation and transmit updated actuator values back toward the simulation. The interface of the `HILInterface` component is illustrated in 4.3.

Figure 4.3: HILInterface component interface.

Stellenbosch University https://scholar.sun.ac.za

The HILInterface exchanges data directly with the Matlab Simulink simulation using UDP packets (see section 3.8 for further details). The device attribute ( ) is used to configure the IP address of the computer running the simulation. The HILInterface contains a UDP helper object that is responsible for sending and receiving of UDP packets.

When the sync event ( ) dispatched by the RateGenerator component is received, reception of emulated sensor values from the simulation is enabled. The received data is continuously accumulated in a buffer until a complete message is detected. The protocol used to encapsulate messages is retained from previous flight control systems - see e.g. [10] for a description. A complete message could contain either emulated GPS or IMU sensor values. Depending on the type of message, a gps_data_update or analog_data_update event ( ) is sent. The GPS and IMU components will receive these events when their execution is invoked and retrieve the updated sensor readings from the HILInterface component using the get_gps_data and get_analog_data facets ( ). The data is maintained in buffers since the GPS and IMU components will not retrieve it at the instant it becomes available. Provision has also been made for the introduction of additional "custom" sensor values, should the need arise, by specifying custom_data_update event and get_custom_data facet ports.

The HILInterface is able to receive/send notifications from/to the Command component through its message_from_command and message_to_command event source ( ) and sink ( ). Currently it only acts on a notification sent from the ground-station to enable HIL mode. The notification to enable HIL mode activates the UDP helper object and prepares it to send and receive data packets from the simulation.

The HILInterface is also responsible to write updated actuator values back to the simulation. It uses the get_tx_sb_timer_data receptacle ( ) to retrieve updated actuator values from the Servo component when it receives the new_tx_sb_timer_data_available event ( ), indicating that new data is available. The data is then written back to the simulation using the UDP helper object.

4.3.5 GPS

The GPS component represents measurements derived from a GPS sensor. The interface of the component is shown in figure 4.4.

Stellenbosch University https://scholar.sun.ac.za

Figure 4.4: GPS component interface.

The implementation of the GPS component is not linked to any particular type of GPS sensor. It expects to retrieve raw GPS measurements through the get_new_gps_data receptacle (). This design was chosen in order to allow the introduction of any type of GPS sensor without having to change the internals of the GPS component. The design also allows it to retrieve emulated GPS measurements from the HILInterface component using its get_gps_data facet (). To introduce a new type of GPS sensor, it will be necessary to create an I/O-type driver component for the specific sensor that provides a get_gps_data facet, which can be linked up to the get_new_gps_data receptacle of the GPS component.

The execution of the GPS component is invoked when it receives an execute event on its receive_execute event sink (). The component is notified of the availability of new GPS measurements through its new_data_available event sink. Updated GPS measurements are then retrieved through the get_new_gps_data receptacle () when the component executes. Depending on the type of measurement received - it could be either position or velocity measurements - calculations and data extractions are performed to update the current position or velocity related data. Once updated calculations have been performed, the component emits the gps_data_updated event () in order to notify other components of the availability of new GPS data. Interested components are able to retrieve the data through the current_position and current_velocity facets () which return a structure filled with relevant position and velocity measurements.

The GPS can receive/send notifications from/to the Command component through its message_from_command and message_to_command event sources () and sinks (). When it receives a receive_timeout_1 event () from the RateGenerator, the component sends updated telemetry measurements back to the ground-station using the message_to_command event source ().

4.3.6 IMU

The IMU component is a container for accelerometer, gyroscope, magnetometer and pressure sensor measurements. The facets, receptacles, event sources and sinks of the component are shown in figure 4.5.

Stellenbosch University https://scholar.sun.ac.za

Figure 4.5: IMU component interface.

Like the `GPS` component, the `IMU` component executes when its `receive_execute` event sink (▷) receives the execute notification from the `RateGenerator` component. It retrieves updated measurements using the `get_new_analog_data` receptacle (☽) once it executes and only after the `new_analog_data_available` event (▷) was received. The new measurements are used to update accelerometer, gyroscope, magnetometer and pressure sensor readings. The measurements are compensated for any offset that may have been introduced during sampling, as this was also done in the original ESL avionics firmware from which the IMU component was ported.

Other interested components are able to retrieve the new sensor data using the facets (○) that have been provided for that purpose:

- The `current_acceleration` facet provides a vector containing updated accelerometer sensor data for the x,y and z axes.
- The `current_magnetization` facet provides a vector containing updated magnetometer sensor data for the x,y and z axes.
- The `current_orientation` facet provides a vector containing updated gyroscopic sensor data for the x,y and z axes.
- The `current_presssure_measurements` facet provides a structure containing updated pressure measurement data such as altitude and absolute, differential pressure measurements.

The `IMU` is also equipped to receive/send notifications from/to the `Command` component through its `message_from_command` and `message_to_command` event sources (□) and sinks (▷). It acts on commands from the ground-station to re-initialize accelerometer, gyroscope and pressure sensor measurements. Updated telemetry measurements are sent back to the ground-station via the `Command` component using the `message_to_command` event source when a notification is received on `receive_timeout_3` sink (▷).

Once the `IMU` sensor data has been updated, the component moves execution off to the next component using its `execute` event source (□).

Stellenbosch University https://scholar.sun.ac.za

4.3.7 Estimator

The `Estimator` component uses accelerometer, gyroscope and magnetometer measurements from the `IMU` component, as well as position and velocity measurements from the `GPS` component in order to estimate the current state of the aircraft. The interface of the `Estimator` component is shown in figure 4.6.

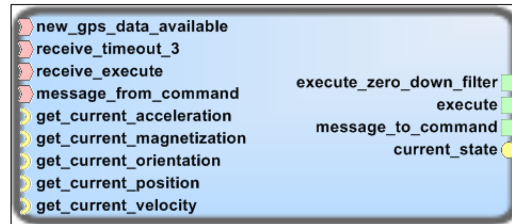


Figure 4.6: Estimator component interface.

As with the other components, the estimators algorithms were ported from existing ESL flight control systems and adapted to function in a component based system. The `Estimator` retrieves updated sensor measurements from `IMU` and `GPS` components when the execution of the component is invoked using its `receive_execute` event sink (). Measurements from the `IMU` are retrieved using the `get_current_acceleration`, `get_current_magnetzation` and `get_current_orientation` receptacles (). These link into the `current_acceleration`, `current_magnetization` and `current_orientation` facets () provided by the `IMU` component. Additionally, position and velocity measurements are retrieved using the `get_current_position` and `get_current_velocity` receptacles which link into `current_position` and `current_velocity` facets of the `GPS` component.

Once the new measurements have been retrieved, the estimator performs updated position, velocity and orientation (angle) state calculations. These are made available to other components through the `current_state` facet ().

Like other passive components, the `Estimator` is also equipped to receive/send notifications from/to the `Command` component through its `message_from_command` and `message_to_command` event sources () and sinks (). It acts on notifications from the ground-station to initialize, enable and disable estimation. The estimator is able to perform full inertial estimation if the ground-station enables this facility in the estimator initialization notification. All other received notifications are ignored. Telemetry measurements are sent back to the ground-station upon reception of a notification on the `receive_timeout_3` event sink ().

Stellenbosch University https://scholar.sun.ac.za

When state estimation is completed, the Estimator moves execution off to the next component using its execute event source (▢).

4.3.8 Controller

The Controller component uses state estimates produced by the Estimator and orientation (gyro), acceleration and pressure measurements from the IMU together with guidance input from a path-planning function to maintain autonomous flight along way-points that have been provided by the ground-station. Figure 4.7 represents the interface of the Controller.

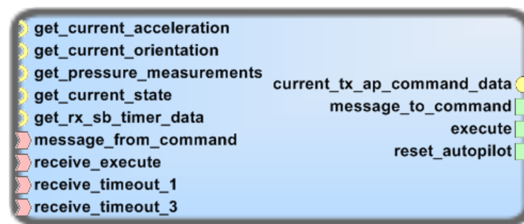


Figure 4.7: Controller component interface.

The controller and path planning algorithms have been ported from existing ESL flight control systems [54]. Four separate inner loop controllers are contained in the Controller component - three *specific acceleration controllers* and a *roll rate controller*. State estimates, orientation, acceleration and pressure measurements that are required as input to the controllers are retrieved using the get_current_state, get_current_orientation, get_current_acceleration and get_pressure_measurements receptacles (▢).

As with the other components, the Controller is also equipped to receive/send notifications from/to the Command component through its message_from_command and message_to_command event sources (▢) and sinks (▢). It acts on a number of notifications and commands from the ground-station. These include commands to enable the autopilot, waypoint navigation and various "inner loop" controller commands that configure control of roll angle, velocity, pitch rate and other aircraft behaviours. It also includes commands to set the desired target values for the controlled quantities.

The Controller retrieves state and sensor measurements from Estimator and IMU when its execution is invoked and the autopilot has been enabled. The various controller updates are then run sequentially, which results in new actuator commands being produced for throttle, aileron, elevator and rudder. The actuator commands are stored until the Servo command retrieves them using the current_tx_ap_command_data facet (▢).

Stellenbosch University https://scholar.sun.ac.za

The Controller sends telemetry related to the functioning of the path planning and navigation algorithms upon reception of a notification on its receive_timeout_1 event sink (). Similarly, it sends telemetry related to the functioning of the various controller algorithms when a notification is received on the receive_timeout_3 event sink.

When the controller algorithm updates have been completed, the Controller pushes execution to the next component using its execute event source ().

4.3.9 Servo

The Servo component takes the actuator commands calculated by the Controller and transforms them into timer values that can be written out to the aircraft actuators in order to effect updates to throttle, aileron, elevator and rudder. The interface of the Servo component is shown in figure 4.8.

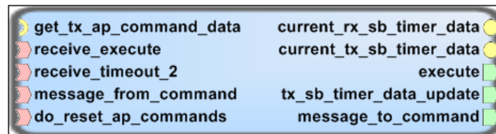


Figure 4.8: Servo component interface.

The Servo component retrieves updated actuator commands when it receives the push to execute via its receive_execute event sink (). The actuator commands are obtained using the get_tx_ap_command_data receptacle () which links into the current_tx_ap_command_data facet () provided by the Controller. The actuator commands are then transformed into timer values using an algorithm that was ported from an existing ESL flight control system. Following this, the Servo notifies interested components of the availability of new actuator timer values using the tx_sb_timer_data_update event (). The notification is picked up by the HILInterface component, which retrieves updated actuator timer values using the current_tx_sb_timer_data facet () provided by the Servo component.

The Servo component is also able to receive/send notifications from/to the Command component through its message_from_command and message_to_command event sources () and sinks (). For its operation, the only events of interest are the enabling/disabling of the autopilot since this affects how the timer values are derived. Should the autopilot not be enabled, then the commands from the safety pilot are transferred directly to the actuators.

Stellenbosch University https://scholar.sun.ac.za

The Servo sends actuator telemetry upon reception of a notification on its receive_timeout_2 event sink (). Since the Servo component is the last component required to execute per update cycle, it does not push execution to another component. It is however easy to extend the component with a execute event source () through the interface definition (IDL) files, should more components be added to the system.

4.3.10 OnBoardComputer

The OnBoardComputer is a component that does not perform functionality that is essential toward maintaining autonomous flight. It exists in order to collect status information on the functioning of the flight control system and report this back to the ground-station via telemetry messages. The interface of the OnBoardComputer is shown in figure 4.9.

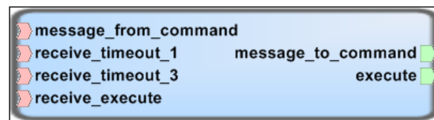


Figure 4.9: OnBoardComputer component interface.

In order to perform its functionality, the OnBoardComputer is equipped to receive/send notifications from/to the Command component through its message_from_command and message_to_command event sources () and sinks ().

Primary telemetry that is collected by this component includes the activation and configuration status of various functions such as autopilot, estimator, and HIL. The secondary telemetry that is gathered contains the voltage levels of the main and backup batteries, as well as the voltage level supplied to the actuator board. Upon reception of notifications on the receive_timeout_1 and receive_timeout_3 event sinks (), the primary and secondary telemetries are sent back to the ground-station using the message_to_command event.

The OnBoardComputer pushes execution to the next component using its execute event source ().

4.3.11 HIL simulation

The HIL simulation setup that has been used in the past at the ESL for most avionics systems involved the use of a distribution board which converted emulated sensor values received serially (USB) from the computer into CAN packets that can be injected into the field bus, as shown in figure 4.10.

Stellenbosch University https://scholar.sun.ac.za

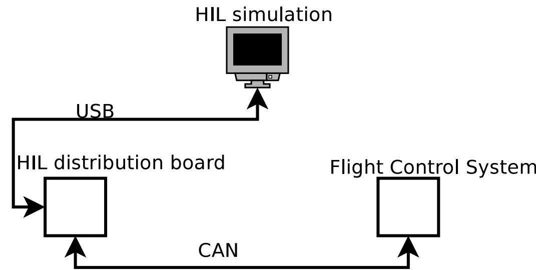


Figure 4.10: Existing HIL simulation setup.

This simulation setup required some modification since the new avionics architecture would be distributed around an Ethernet switch. The new architecture enabled the Matlab HIL simulation software to write out the emulated sensor values via Ethernet to the processor board that is being tested, bypassing the need for a distribution board altogether, as shown in figure 4.11.

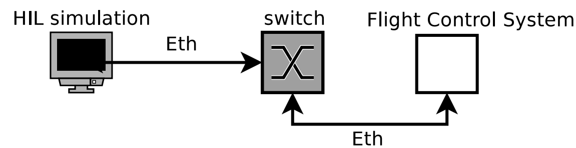


Figure 4.11: New HIL simulation setup.

Previously, values were written out serially (the distribution board is equipped with an FTDI chip converting RS-232 to USB) to the HIL distribution board from the Matlab simulation by hooking into the `mdlOutputs` method, which Simulink calls during every simulation time step. The serial output routines were replaced in the `mdlOutputs` method with new functions writing out UDP packets toward the IP address that had been configured.

4.4 Conclusion

In this chapter, the design of the individual hardware and software functions that are required in the new avionics architecture were described. In chapter 5 it is shown how the functions were integrated to form a complete system.

Stellenbosch University https://scholar.sun.ac.za

Chapter 5

System Integration

In chapter 4 the individual hardware and software functions of the flight control system architecture were designed. In this chapter, the functions are put together to form a complete functioning system. The majority of the effort required was on the software side. On the hardware side, COTS, prototyping and existing actuator boards have been used thereby avoiding any custom printed circuit board development.

5.1 Hardware

Figure 3.9 showed the hardware block diagram of the flight control system as it would be used for flight tests. Since this project relied on HIL simulation testing, the final, tested hardware part of the system implementation is illustrated in the block diagram of figure 5.1.

Stellenbosch University https://scholar.sun.ac.za

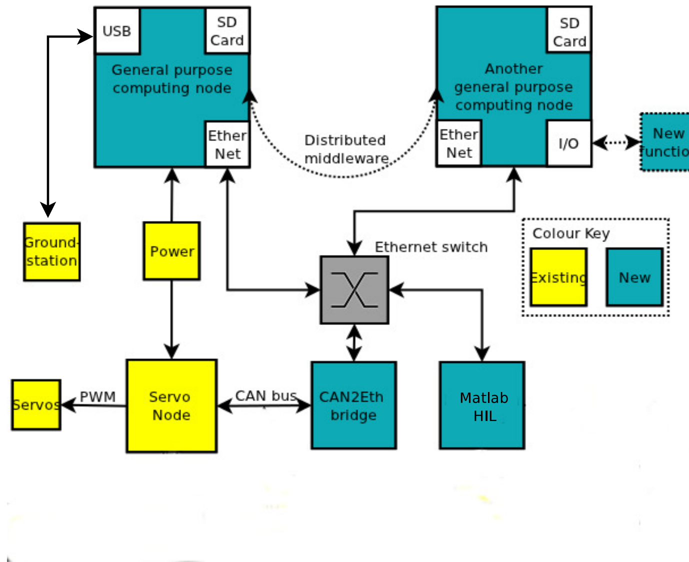


Figure 5.1: HIL tested hardware block diagram of the flight control system.

The `PREEMPT_RT` patched Linux-kernel based OS that was created using Yocto recipes was transferred to a SD-card and used as the bootable medium for the Gumstix SBC computing node.

5.2 Software

Integrating the various components into a complete function system is achieved through a number of steps prescribed by the OMG Deployment and Configuration standard [49]. The standard prescribes how to install, configure and launch a component-based software application. The TAO/CIAO distribution middleware that was used for this project includes a standards compliant implementation, known as the Deployment and Configuration Engine (DaNCE) that aids in executing the process [50]. The process will be explained using the `RateGenerator` component as an example, however certain steps may need to be repeated for all components in the system. For convenience, the component is shown again in figure 5.2.

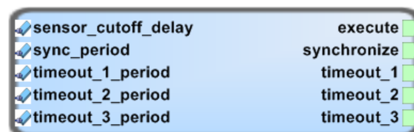


Figure 5.2: Rate Generator component interface.

Stellenbosch University https://scholar.sun.ac.za

5.2.1 Preparation

Once the core functionality has been implemented, the software components have to be packaged in a suitable manner. To this end, each "software artefact" of the component was compiled into a shared library which represents the deployable implementation of a component. The main artefacts of a component include:

- the user-implemented core functionality of the component, also referred to as the executor,
- a compiler-generated servant of the component, which houses the remote method invocation functionality associated with the component, and,
- a compiler-generated stub, which represents the public interface of the component as seen by other components.

5.2.2 Installation

The configuration and deployment standard makes provision for the packaged software components to be stored in a repository from where they can be retrieved during the installation process. The repository does not have to located on the same system where the software will execute. For this project, the decision was made to keep it simple - installation consisted of transferring all required shared libraries, associated binaries and start-up scripts to a suitable location on the target host.

5.2.3 Configuration

All the software artefacts required to enable complete functionality of the system have to be registered with the configuration and deployment tool of the distribution middleware, so that they can be located during deployment of the components into the middleware server runtime. This is achieved by listing them in a configuration and deployment XML file, a minimal example of which is shown in the listing below:

```
1  <Deployment:DeploymentPlan>
2  <artifact xmi:id="IO-Rate_Generator_exec">
3      <name>Rate_Generator_exec</name>
4      <source/>
5      <node/>
6      <location>Rate_Generator_exec</location>
7  </artifact>
8  <artifact xmi:id="IO-Rate_Generator_svnt">
9      <name>Rate_Generator_svnt</name>
10     <source />
```

Stellenbosch University https://scholar.sun.ac.za


```
11    <node />
12    <location>Rate_Generator_svnt</location>
13 </artifact>
14 </Deployment:DeploymentPlan>
```

The listing only shows the configuration of RateGenerator artefacts, however all artefacts of each developed component required for a complete system have to be registered. The <name> child node in the listing maps to the shared library that was created for each of the component artefacts. In the case of the RateGenerator, Rate_Generator_exec in the listing above refers to the previously described executor, while Rate_Generator_svnt refers to the servant.

The next step is to describe the implementation of the component. This involves listing all software artefacts that have been previously declared which are relevant to the component. Additionally, entry points are also declared that will be used by the deployment tool to launch the component - these are the execParameter nodes in the listing. In the case of a C++ implementation, the home factory entry point is the name of the method that returns a pointer to a constructed executor - i.e. that part of the component which implements the core functionality. Similarly, the ServantEntryPoint returns a pointer to the constructed servant, which is responsible for handling any remote method invocations on the component. For the purposes of brevity, some details have been omitted where ellipses (. . .) are shown.

```
1  <Deployment:DeploymentPlan>
2  <implementation xmi:id="IO-Rate_Generator-mdd">
3     <name>IO-Rate_Generator-mdd</name>
4     <source />
5     <artifact xmi:idref="IO-Rate_Generator_svnt" />
6     <artifact xmi:idref="IO-Rate_Generator_exec" />
7     <execParameter>
8        <name>home factory</name>
9        ...
10    </execParameter>
11    <execParameter>
12       <name>edu.vanderbilt.dre.CIAO.ServantEntrypoint</name>
13       ...
14    </execParameter>
15 </implementation>
16 </Deployment:DeploymentPlan>
```

The description of a component implementation is followed by configuring properties and attributes that could be of use to a running instance of the

Stellenbosch University https://scholar.sun.ac.za

component, as shown in the following listing.

```
<Deployment:DeploymentPlan>
<instance xmi:id="IO-Rate_Generator-idd">
    <name>IO-Rate_Generator-idd</name>
    <node>Rate_GeneratorNode</node>
    <source />
    <implementation xmi:idref="IO-Rate_Generator-mdd" />
    <configProperty>
    ...
    </configProperty>
</instance>
</Deployment:DeploymentPlan>
```

For the `RateGenerator` component, the attributes configuring the various time-out periods need to be configured. An example configuration of the `sensor_cutoff_delay` time-out period is given in the listing below:

```
<configProperty>
    <name>sensor_cutoff_delay</name>
    <value>
        <type>
            <kind>tk_long</kind>
        </type>
        <value>
            <long>3000</long>
        </value>
    </value>
</configProperty>
```

The reactor object used to implement the timers (see 4.3.2) expects input to be provided in micro-seconds, therefore a value of 3000 is provided to obtain a $3ms$ time-out. Any other configurable parameters of interest could be defined in this section.

The final configuration that is required to launch the complete application is to appropriately connect the various interfaces of one component to another. Two types of connections exist: event sources to sinks, and facets to receptacles. The listing below shows how the`timeout_1` event source of the `RateGenerator` component is connected to the `receive_timeout_1` event sink of the `GPS` component.

```
</Deployment:DeploymentPlan>
<connection>
    <name>Rate_Generator_to_GPS_timeout_1_event</name>
    <internalEndpoint>
```

Stellenbosch University https://scholar.sun.ac.za


```
5        <portName>timeout_1</portName>
6        <provider>false</provider>
7        <kind>EventPublisher</kind>
8        <instance xmi:idref="IO-Rate_Generator-comp-idd" />
9    </internalEndpoint>
10   <internalEndpoint>
11       <portName>receive_timeout_1</portName>
12       <provider>true</provider>
13       <kind>EventConsumer</kind>
14       <instance xmi:idref="Sensors-GPS-comp-idd" />
15   </internalEndpoint>
16 </connection>
17 </Deployment:DeploymentPlan>
```

The syntax required to connect a receptacle to a facet is very similar. Since the `RateGenerator` component has no facet or receptacle to connect, the example below shows how the `current_acceleration` facet of the `IMU` component is connected to the `get_current_acceleration` receptacle of the `Estimator` component.

```
1  </Deployment:DeploymentPlan>
2  <connection>
3     <name>IMU_to_Estimator_acceleration_facet</name>
4     <internalEndpoint>
5        <portName>current_acceleration</portName>
6        <provider>true</provider>
7        <kind>Facet</kind>
8        <instance xmi:idref="Sensors-IMU-comp-idd" />
9     </internalEndpoint>
10    <internalEndpoint>
11       <portName>get_current_acceleration</portName>
12       <provider>false</provider>
13       <kind>SimplexReceptacle</kind>
14       <instance xmi:idref="Navigation-Estimator-comp-idd" />
15    </internalEndpoint>
16 </connection>
17 </Deployment:DeploymentPlan>
```

5.2.4 Launching

The launch process brings the flight control system into a executing state by using the XML based configuration and deployment file described above to reference all required resources. Initially a server run-time environment (also referred to as node) that has its own associated memory and processing resources is first prepared per component. The run-time environment is provided

Stellenbosch University https://scholar.sun.ac.za

by the deployment and configuration tool, which is part of the distribution middleware. Any additional services that are required by the application, such as a naming service that could be used by the components to locate each other, are also started up.

An execution manager, also part of the distribution middleware, then instantiates each component into its own run-time environment. During this process, component attribute values are initialized using the values provided in the configuration file. Event sources/sinks as well as facets/receptacles are also connected as per the configuration file.

Finally, a small driver program is used to activate the reactor object in the `RateGenerator` component. Once the reactor object is activated, the various timers are enabled and the system is put into an executing state. This design was chosen in order to be able to control the commencement of execution in the system.

5.2.5 Model Driven Design

Alternatives have been developed to writing configuration and deployment files by hand, since the activity may become tedious. It is also possible that the task can become error-prone as the number of components in a system increases, which will likely result in difficult-to-debug problems occurring during system configuration and/or activation.

To aid with this difficulty, model driven design and architecture tools, first introduced in section 2.6 have been developed. They allow component interfaces to be designed visually, as was shown in chapter 4. Iterative adjustments that often occur during development of a system are also more intuitive on a visual model than a XML file. Once the component interface has been modelled, interconnections between ports can be added, as shown in figure 5.3 using the example of the `IMU` and `Controller` component.

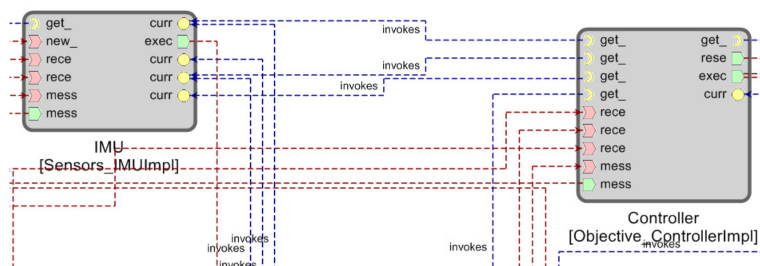


Figure 5.3: Visual modelling of port connection.

Stellenbosch University https://scholar.sun.ac.za

The point of visual modelling of components and interfaces is that it enables software artefacts, such as IDL and deployment files, that are required to develop and run distributed systems, to be generated from the model. Freely available tools, such as the Generic Modelling Environment, together with domain specific modelling languages and interpreters that have been adapted for component-based development, are available that provide this functionality [55].

5.3 Conclusion

In this chapter, it was shown how the individual hardware and software components that comprise the flight control system were integrated into a complete unit. In chapter 6 the results of system testing to verify correct functionality of the complete system is presented.

Stellenbosch University https://scholar.sun.ac.za

Chapter 6

System Verification

In chapter 5 it was shown how the individual hardware and software functions were integrated and combined into a complete flight control system. New flight computing hardware was introduced, while CAN-Ethernet functionality was also designed to enable the reuse of existing actuator hardware. The system was built in a distributed fashion around a COTS Ethernet switch to enable extendibility. The component-based avionics system application was designed to run on a real-time OS to ensure deterministic behaviour.

In this chapter, results of experiments are presented that were performed in order to quantitatively and qualitatively ascertain how well the system is performing. Tests for the CAN-Ethernet gateway, real-time OS and Ethernet switch are described in sections 6.1, 6.2 and 6.3.

As the flight control logic implemented in the software components was ported from a previous system in use at the ESL [54], an experiment was completed in which both existing and new systems were run in parallel and telemetry was used to determine functional equivalence. Some modifications to the newly developed avionics system were required to enable direct comparison with an existing system, which, together with the results, are described in section 6.4. The testing was performed using a Matlab Simulink HIL simulation which was previously developed at the ESL [10].

Testing of the complete avionics system in its final form is documented in section 6.5. Testing for the system was also completed using the Matlab HIL simulation environment.

Since a goal of this thesis was to enable extendibility of the flight control unit, the system was also tested with a setup in which component functions were distributed over two host platforms. The results of the experiment are documented in subsection 6.6.

Stellenbosch University https://scholar.sun.ac.za

The chapter concludes with an overview of all the results in 6.7.

6.1 CAN-Ethernet Gateway

The correct operation of the CAN-Ethernet Gateway was confirmed by running successful HIL simulation tests using the gateway, as described in section 6.5. However, the amount of latency introduced by the gateway was also of interest, since it is a new function that has been inserted into a distributed architecture.

To quantitatively measure the latency proved to be cumbersome, since it was difficult to track the ingress of a particular CAN packet up until the point at which it exits as a UDP packet and vice versa. Instead, to qualitatively evaluate the performance of the gateway, two tests were performed: in the first test, two gateways were inserted into a HIL simulation setup of an existing flight control system, as shown in figure 6.1. The HIL simulation was run successfully in this test, despite the presence of one more CAN-Ethernet translation step in the network than would normally be used in a system.

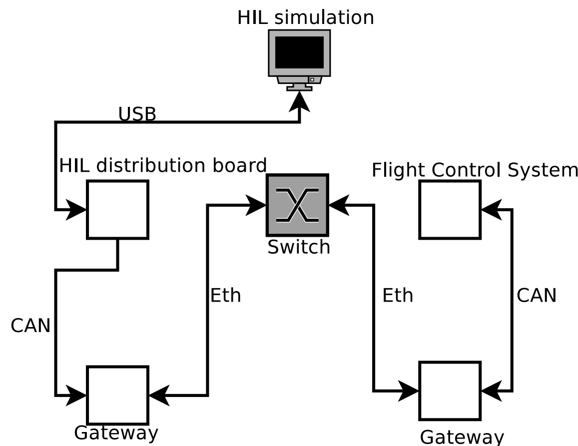


Figure 6.1: CAN-Ethernet gateway test setup.

In the second test, a separate CAN device was programmed to transmit at a speed of 800kbps, which is the speed at which the existing sensor and actuator devices operate at. CAN packets were then continuously sent in order to consume the available bandwidth on the CAN interface. The data in the CAN packets was populated with numbers that were increasing in a loop until a maximum was reached, at which point it was reset. The CAN-Ethernet gateway was set-up to count the number of lost packets as soon as it detected collisions of CAN packets and write this statistic to the prototype board screen. During an hour test, no lost CAN packets were detected. Correct number sequencing was also found in the received UDP packets.

Stellenbosch University https://scholar.sun.ac.za

6.2 Real-time Operation

6.2.1 RTOS

Although it has been shown that a PREEMPT_RT patched Linux kernel can be used successfully in hard real-time control applications [42], this depends to a degree on the actual hardware and device drivers that are in use. Tasks running on a patched Linux kernel could still incur execution latencies due to hardware generated interrupts or OS generated software signals [41].

To characterise this possibility, tests were performed on the Gumstix computing hardware used for this project in order to measure the latency between when a task is first requested and when it starts executing, as illustrated in figure 6.2.

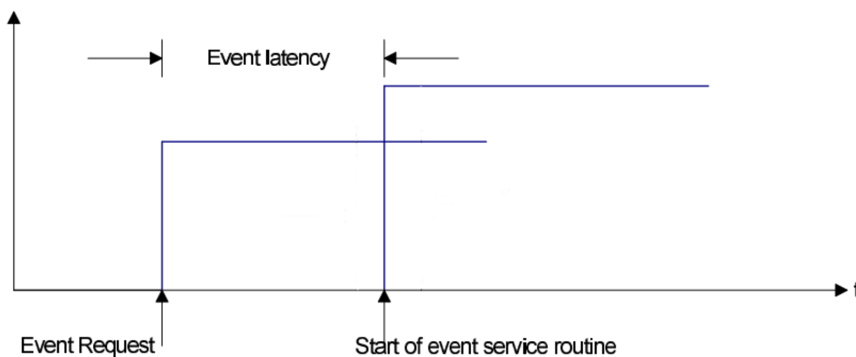


Figure 6.2: Event latency.

The test was performed using the `cyclictest` utility that is part of the PREEMPT_RT patch. A 1000 μs periodic event was repeated to obtain 100000000 samples - approximately 24 hours running time. The test was repeated for a lightly-loaded as well as heavily-loaded system. For the heavy-load scenario, the CPU was kept busy by invoking a continuous loop with a basic arithmetic operation. The CPU load was shown to be close to 100% at the start of the test.

Stellenbosch University https://scholar.sun.ac.za

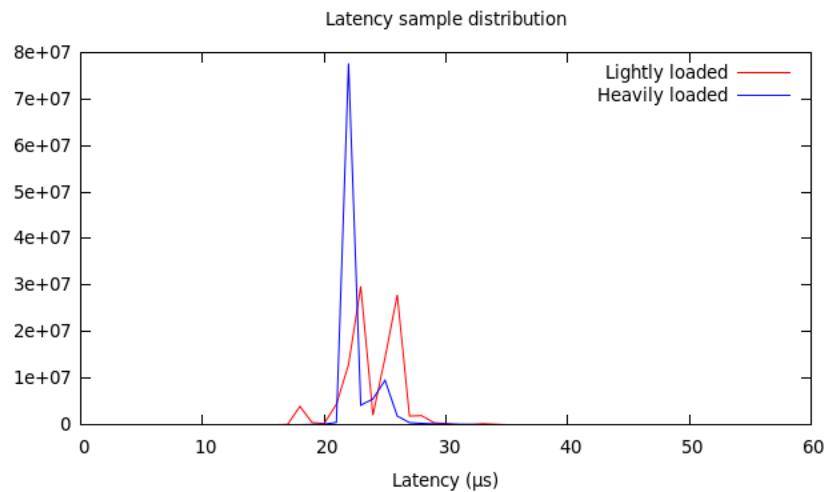


Figure 6.3: Latency distribution graph.

The average latency was found to be 23 μs on a lightly loaded system, with the highest limit being recorded at 53 μs. On a loaded system the average latency was calculated to be at 22 μs, while the highest latency samples were recorded at 60 μs. Figure 6.3 illustrates the distribution in latencies for both systems.

6.2.2 Flight Control System

Confirming the capability of the OS is the first aspect to verifying real-time operation of the system. The ported flight control software also needs to be shown to conform with the timing requirements as described in subsection 3.4.2. Two aspects were investigated:

- total algorithm running time should be completed within a $20ms$ interval.
- execution events should be triggered precisely every $20ms$.

Both characteristics are required to ensure that the avionics system produces actuator output timeously at the required rate in order to maintain stable flight. The algorithm execution time was calculated by measuring the delta between the start time of the first executed component and the end time of the last executed component. Figure 6.4 shows the result of this. As can be seen, the algorithm generally completed within $13ms$, which fits well within the $20ms$ budget.

Stellenbosch University https://scholar.sun.ac.za

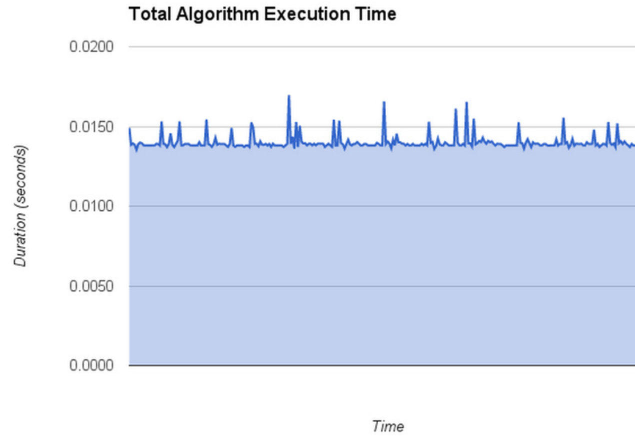


Figure 6.4: Total algorithm execution time.

The control algorithm execution interval was measured by repeatedly logging the exact start time at which the first `execute` event was triggered and then calculating the delta of successive iterations. Figure 6.5 shows the result, and as can be seen the system generally triggered an `execute` event every $20ms$. However, a peaks and troughs pattern could also be observed in the graph that were spaced $0.5s$ apart.

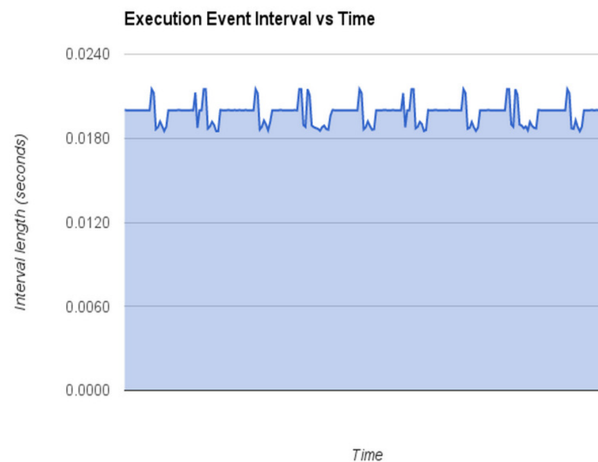


Figure 6.5: Execution event interval.

Upon closer investigation, it was found that the time-out events to trigger the sending of telemetry data were occurring at regular $0.5s$ intervals precisely at the same time an `execute` event was triggered. Once the time-out events were re-scheduled to not occur at precisely the same time as the `execute`

Stellenbosch University https://scholar.sun.ac.za

event, the execution interval triggered every 20ms after a short initial "settling period", as shown in figure 6.6.

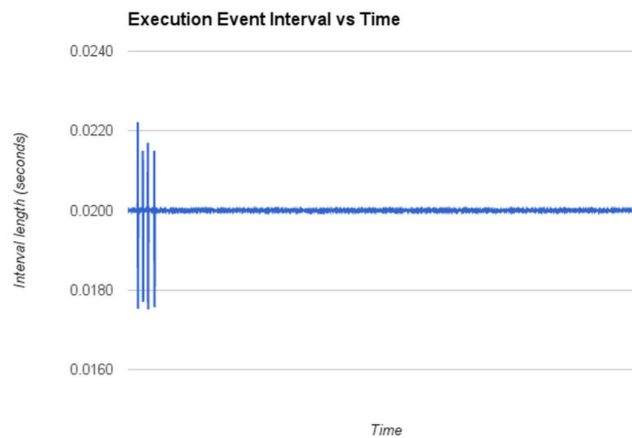


Figure 6.6: Execution event interval after adjustment.

6.3 Ethernet Switch

The distribution test described in the introduction to this chapter also offered the possibility to ascertain the impact on flight performance of the Ethernet switch in the architecture. The distribution of components over two hosts would result in inter-host control and data flow through the switch, thereby providing the chance to qualitatively observe the performance of the switch.

6.4 Comparison

6.4.1 Method

The algorithms of the new avionics system were based on, and ported from an existing system that was previously developed and supplied by the ESL [54]. To determine functional equivalence of the existing and new systems, a method needed to be found that would allow both systems to be run at the same time with the same input data. The response of both systems to the same input data could then be recorded in the telemetry measurements and compared.

A number of modifications to the new avionics system were performed that enabled it to be run in tandem with the existing system while a MATLAB Simulink based HIL simulation was performed. The new avionics system was hosted on a computer running a Linux-kernel based OS, while the HIL simulation was run in a virtual machine hosted on the same computer. The

Stellenbosch University https://scholar.sun.ac.za

HIL simulation exchanged emulated sensor values via a USB-connected HIL distribution board [10] with the existing system. The distribution board acted as a USB-to-CAN gateway, injecting the emulated sensor values received from the simulation on the USB interface into the CAN bus where the existing flight control system was able to retrieve them.

A modification to the `HILInterface` component was required in order to enable it to simultaneously receive the same emulated sensor values from the simulation as the existing system. The modification entailed replacing the object in the `HILInterface` that exchanged data with a simulation via UDP with a new object that extracted data from the same USB interface as was being used by the existing system. The object encapsulated a facility in the Linux kernel known as *usbmon*, which exposes the I/O activity of a USB bus as a text file under the */sys/kernel* file-system and also provides for the data to be read in a binary format from a character device called */dev/usbmonN*, N being the usb bus number that is traced. This facility enabled the `HILInterface` to sniff the USB data exchange between the existing system and the simulation for emulated sensor values and copying these over as they were being sent. This allowed both the existing as well as the new system to run simultaneously off the same emulated sensor values. The existing system then fed back updated actuator positions into the HIL simulation, since the simulation expects to receive updates from one system only. The complete setup is show in figure 6.7.

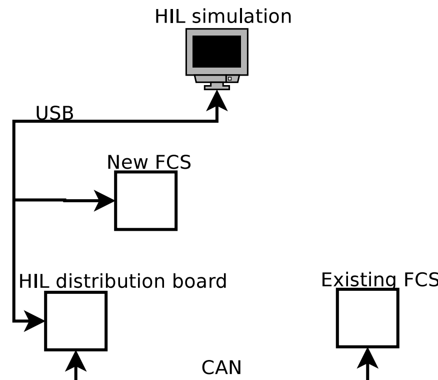


Figure 6.7: Setup for simultaneous operation of both flight control systems.

Since both systems needed to have the same frequency of operations, the timing of the new flight control system was slaved to that of the existing system. This was achieved by removing the `RateGenerator` component and instead basing the timing on packets sent from the Matlab simulation to the existing flight control system. The `HILInterface` started execution of relevant components as soon as it detected appropriate data on the USB interface

Stellenbosch University https://scholar.sun.ac.za

indicating the start of a new sync period (see figure 3.7 for a reminder of control and data flow timing).

To enable reception of ground-station messages, the `Command` component was modified in a similar manner to the `HILInterface`, allowing it to monitor the serial communication between the ground-station and the existing flight control system for any relevant notifications and commands being sent. This also allowed the `Command` component to copy the existing telemetry messages, which turned out to be convenient for comparison purposes.

For the MATLAB Simulink based HIL simulation, four way-points were programmed into the system from the ground-station to form a lap. The runway coincided with the last way-point and was located at 34°2′47.778” S latitude and 18°44′25.072” E longitude. The remaining way-points were spaced 500m apart in a square, as shown in figure 6.8 and table 6.1. All way-points were located 160m above ground. The desired airspeed for the aircraft was set to 22m/s.

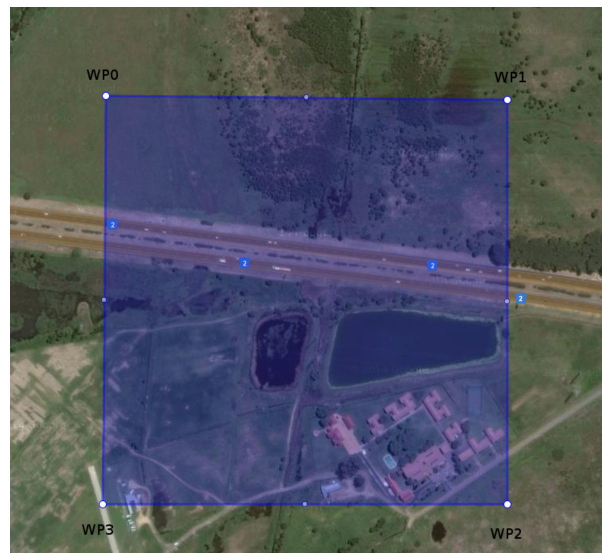


Figure 6.8: Way-points that were used during the simulation.

Stellenbosch University https://scholar.sun.ac.za

Waypoint	North Offset	East Offset	Altitude	Heading	Airspeed
WP0	500m	0m	160m	0°	22m/s
WP1	500m	500m	160m	0°	22m/s
WP2	0m	500m	160m	180°	22m/s
WP3	0m	0m	160m	270°	22m/s

Table 6.1: Way-point details.

Since an RC controller was not available during the development of the system, take-off of the aircraft was achieved by introducing a switch into the Matlab Simulink model that, when toggled to the on-position, enabled maximum throttle. This allowed the aircraft to take-off and gain a measure of altitude. After aircraft take-off, maximum throttle was disengaged while the autopilot was activated in order to allow the system to fly autonomously between the four way-points in a clock-wise manner. Since the aircraft was not in the desired position directly after take-off, the autopilot initially manoeuvred around the first way-point until the correct altitude was achieved and then proceeded to autonomously fly through the remaining way-points, repeating the lap until it was dis-engaged. The results of the telemetry comparisons for various components/functions are presented in the subsequent paragraphs, while the results are discussed in subsection 6.4.6.

6.4.2 Sensors

6.4.2.1 GPS

The GPS component telemetry of position and velocity measurements between existing and new flight control systems indicated good correlation. Figures 6.9, 6.10 illustrate the correlation in GPS reported latitude and longitude, while figures 6.11, 6.12 show the same for north and east velocity.

Stellenbosch University https://scholar.sun.ac.za

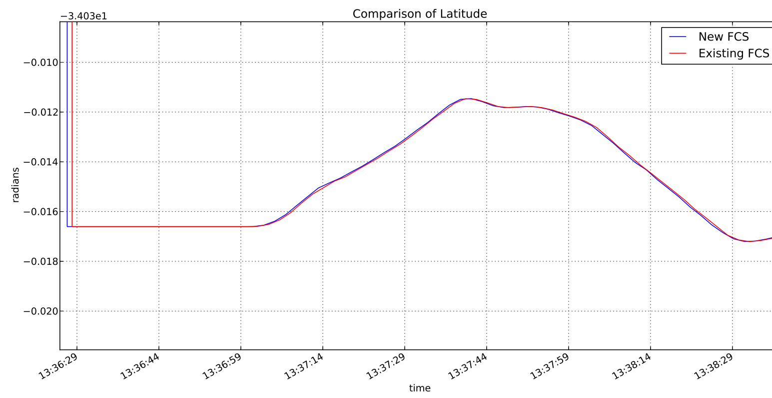


Figure 6.9: Latitude comparison.

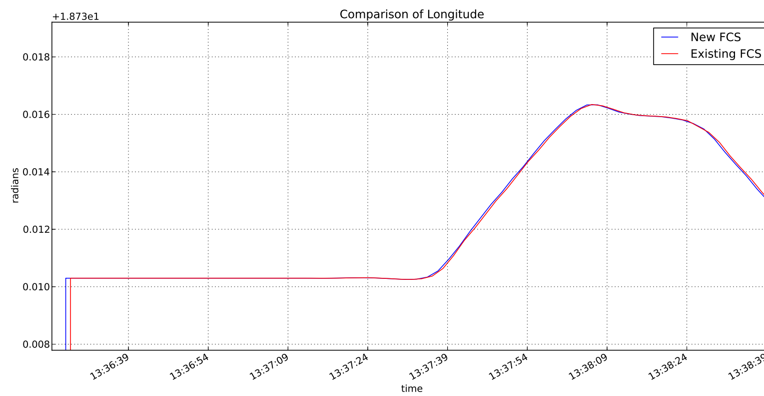


Figure 6.10: Longitude comparison.

Stellenbosch University https://scholar.sun.ac.za

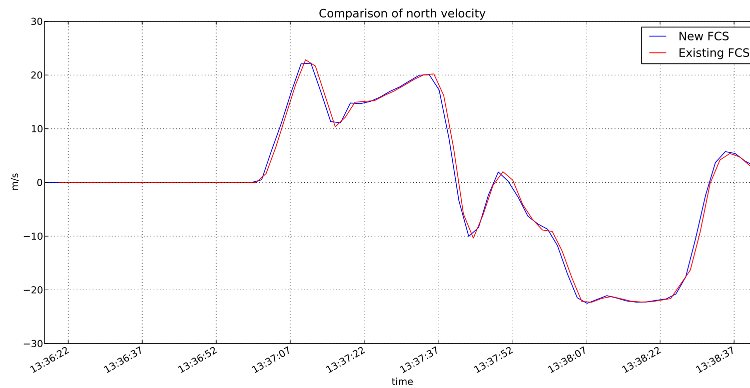


Figure 6.11: GPS north velocity comparison.

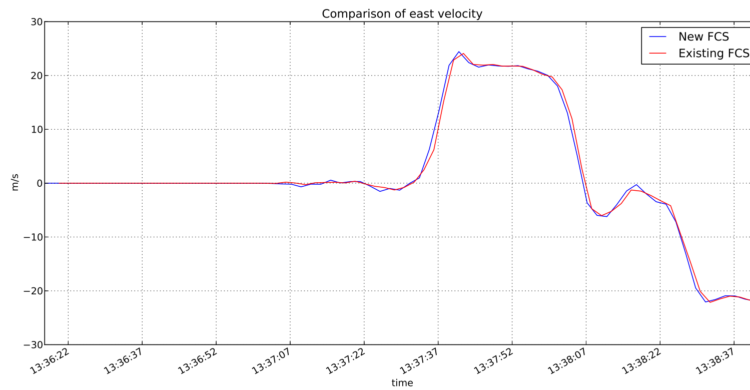


Figure 6.12: GPS east velocity comparison.

6.4.2.2 IMU

As with the `GPS` component, the correlation in `IMU` sensor measurements between the new and existing system was good. Figures 6.13 - 6.15 show the similarity in gyroscope readings, while figures 6.16 and 6.17 indicate that the pitot-static airspeed and altitude telemetry readings from the pressure-meter were also comparable.

Stellenbosch University https://scholar.sun.ac.za

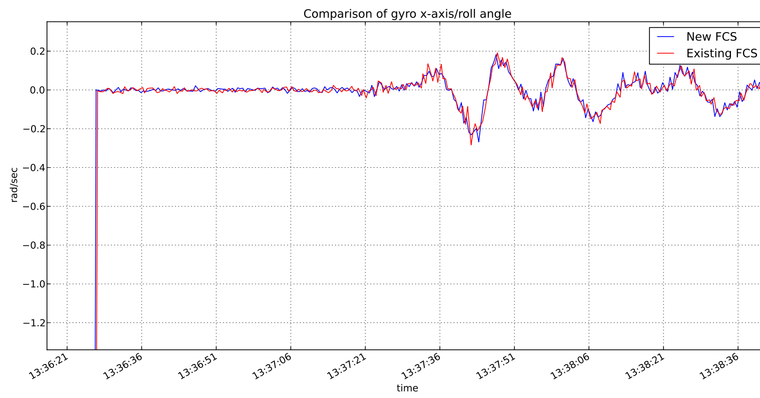


Figure 6.13: Gyro x-axis/roll angle comparison.

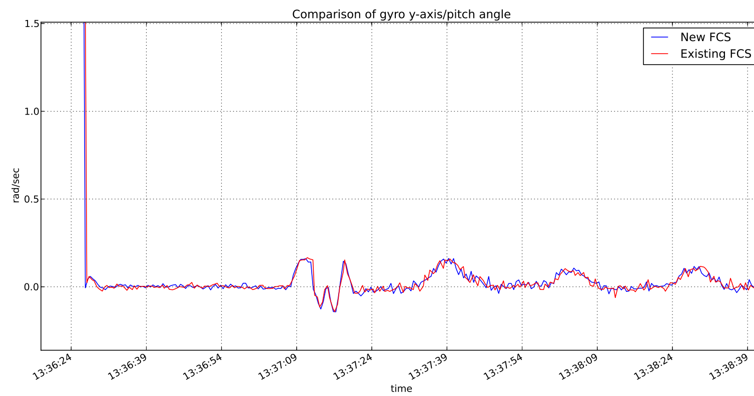


Figure 6.14: Gyro y-axis/pitch angle comparison.

Stellenbosch University https://scholar.sun.ac.za

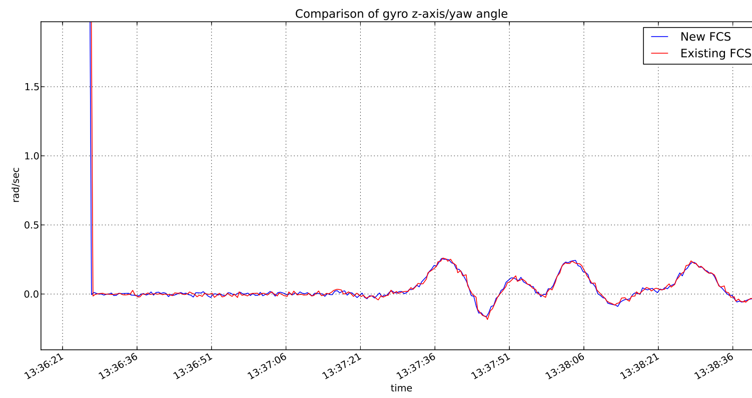


Figure 6.15: Gyro z-axis/yaw angle comparison.

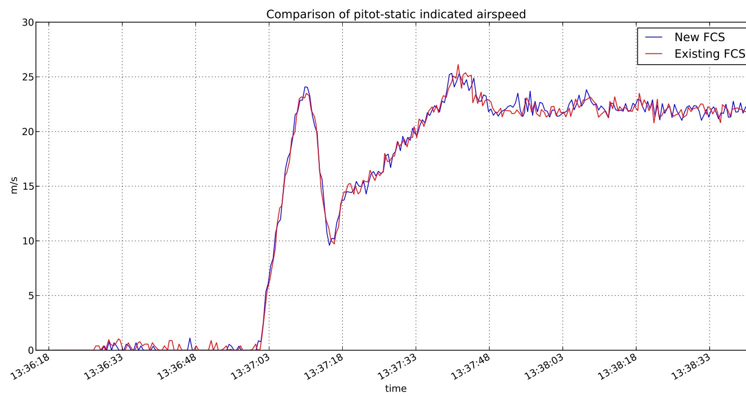


Figure 6.16: Pitot-static indicated airspeed.

Stellenbosch University https://scholar.sun.ac.za

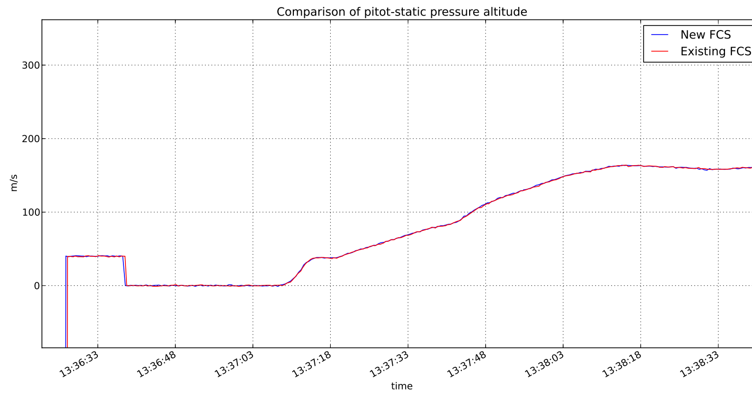


Figure 6.17: Pitot-static pressure altitude.

6.4.3 Estimator

The `Estimator` telemetry readings for the position and velocity estimates shown in figures 6.18 - 6.21 were similar between existing and new flight control systems. State estimation of position and velocity also correlated to a high degree with `GPS` telemetry measurements, as shown previously in figures 6.9 - 6.12. However, the state estimates for roll, pitch and yaw angles, as shown in figures 6.22 to 6.24, exhibit small but discernible differences.

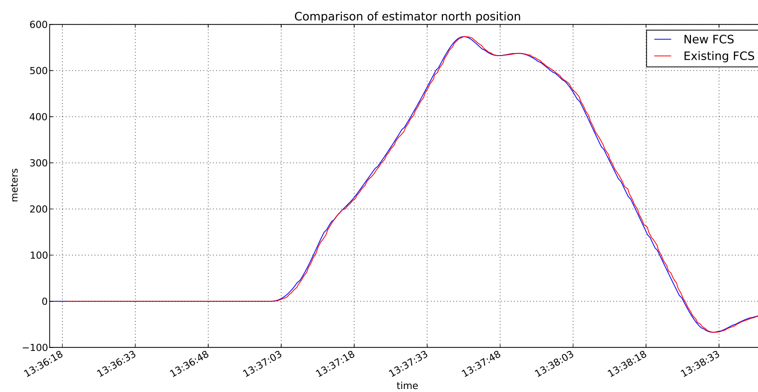


Figure 6.18: Estimator north position comparison.

Stellenbosch University https://scholar.sun.ac.za

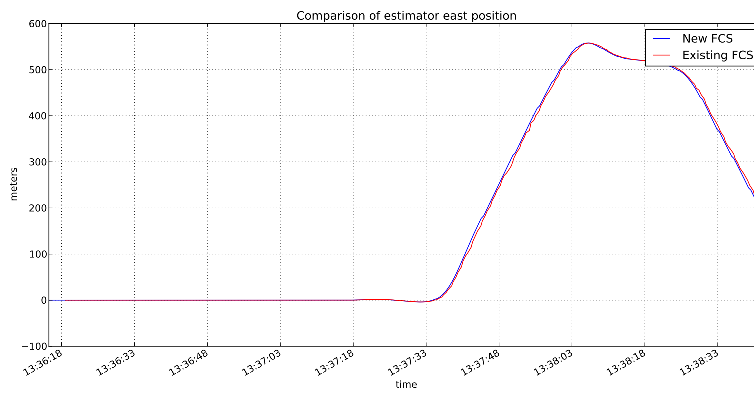


Figure 6.19: Estimator east position comparison.

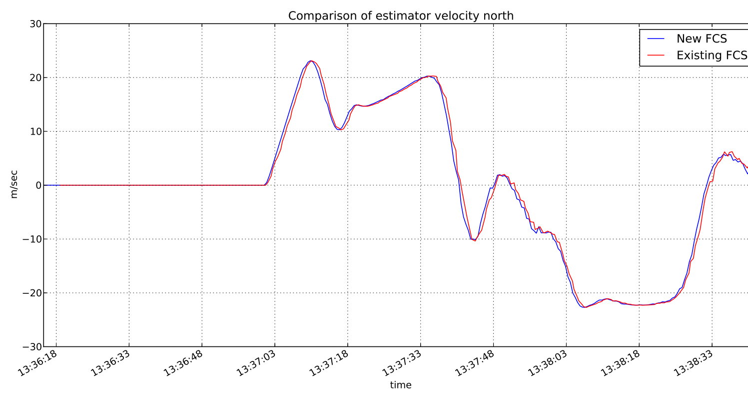


Figure 6.20: Estimator north velocity comparison.

Stellenbosch University https://scholar.sun.ac.za

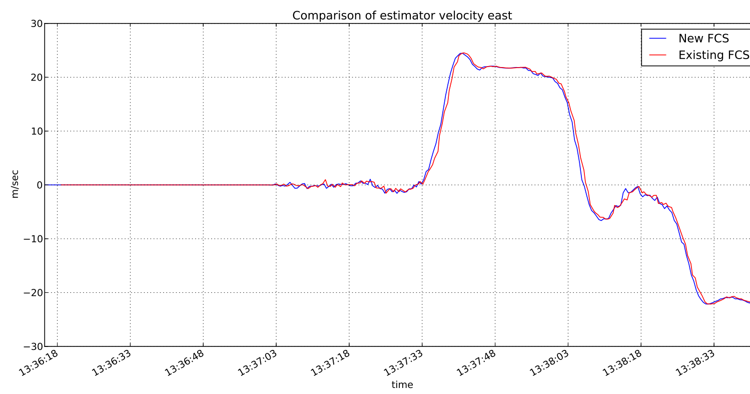


Figure 6.21: Estimator east velocity comparison.

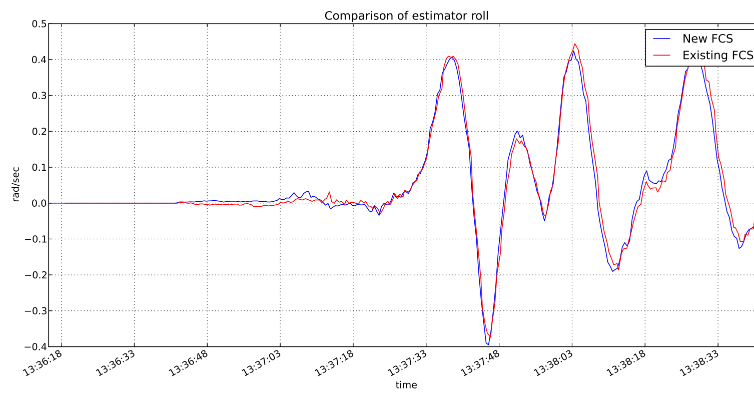


Figure 6.22: Estimator roll angle comparison.

Stellenbosch University https://scholar.sun.ac.za

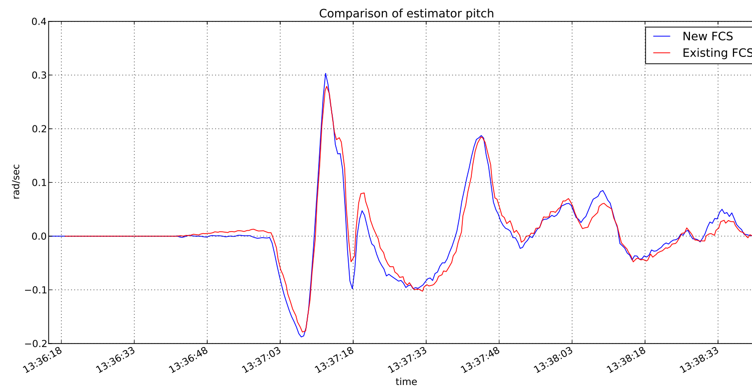


Figure 6.23: Estimator pitch angle comparison.

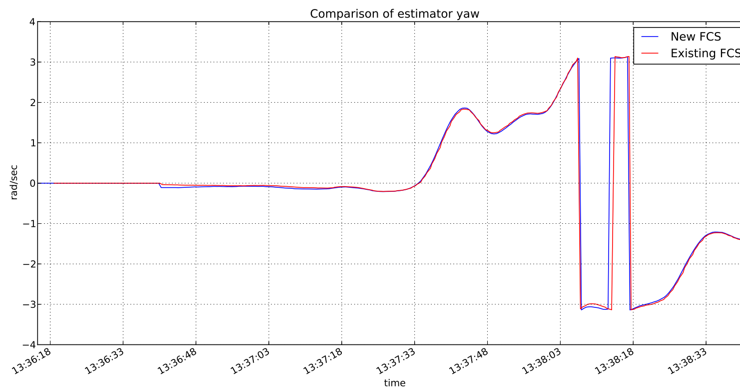


Figure 6.24: Estimator yaw angle comparison.

6.4.4 Controller

Small differences were noted in some of the `Controller` component telemetry measurements. While the telemetry of some of the controller references were similar between existing and new flight controllers, differences could be seen in the bank angle reference (figure 6.25), horizontal acceleration reference (figure 6.26) and time at which navigated waypoints were reached (figure 6.27.

Stellenbosch University https://scholar.sun.ac.za

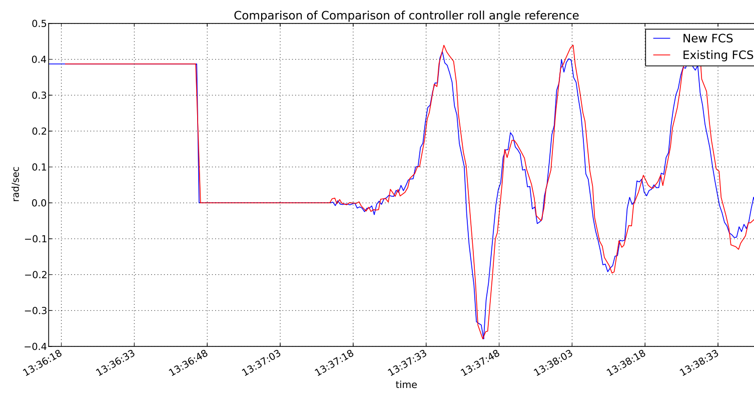


Figure 6.25: Controller bank angle reference comparison.

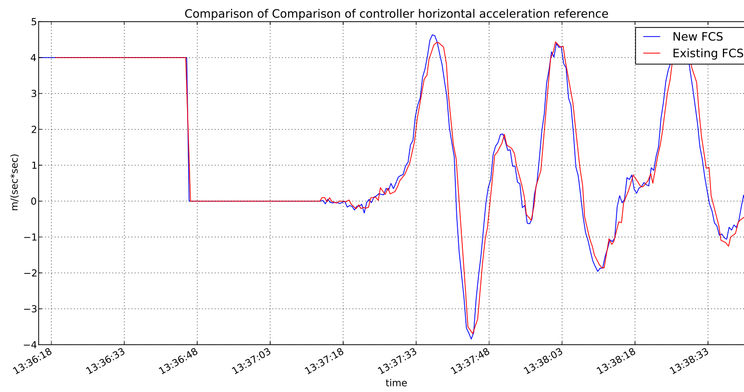


Figure 6.26: Controller horizontal acceleration reference comparison.

Stellenbosch University https://scholar.sun.ac.za

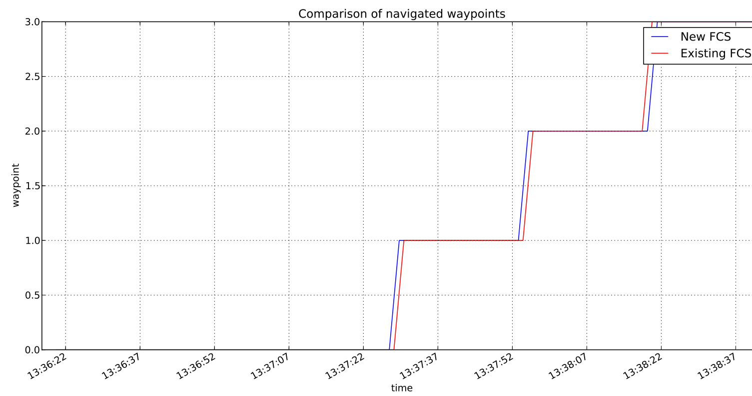


Figure 6.27: Comparison of navigated waypoints.

6.4.5 Servo

The `Servo` telemetry readings, although being similar between existing and new flight controllers, exhibited a number differences, especially on the aileron, elevator and rudder commands, as can be seen in figures 6.28 - 6.31.

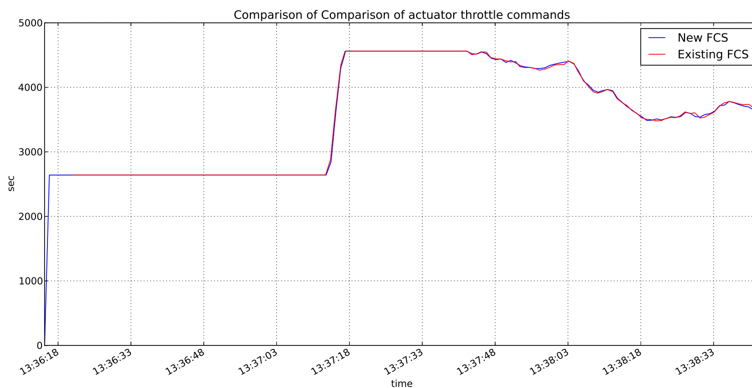


Figure 6.28: Comparison of actuator throttle commands.

Stellenbosch University https://scholar.sun.ac.za

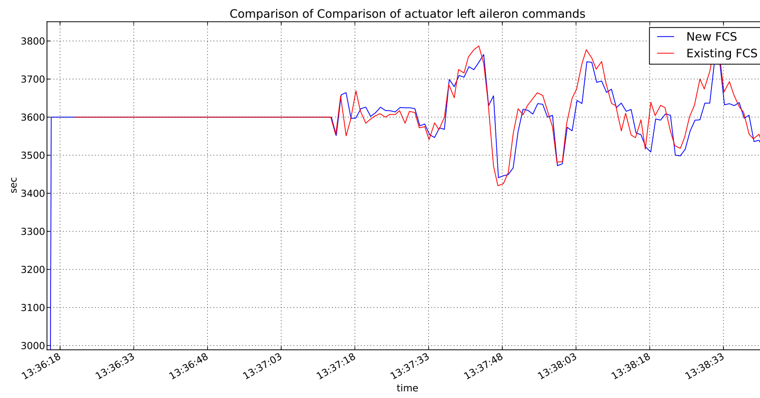


Figure 6.29: Comparison of left aileron commands.

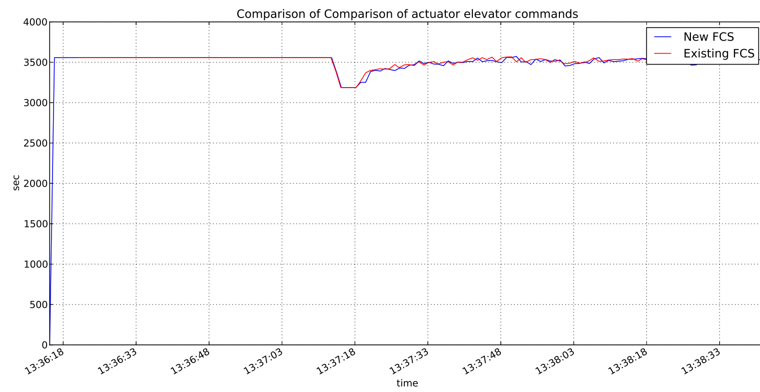


Figure 6.30: Comparison of actuator elevator commands.

Stellenbosch University https://scholar.sun.ac.za

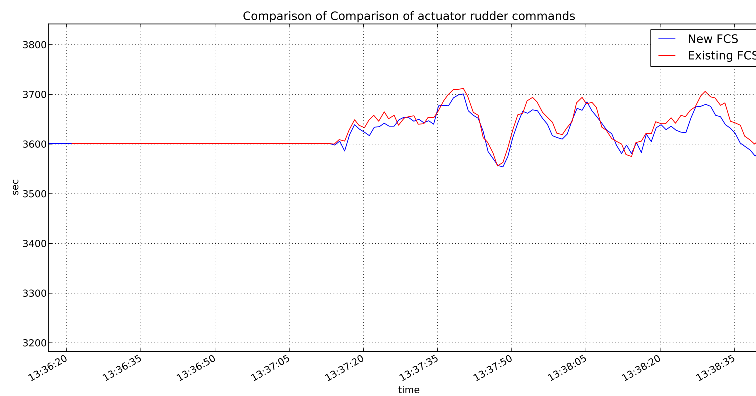


Figure 6.31: Comparison of actuator rudder commands.

6.4.6 Discussion

Since the controller algorithms for the new avionics were ported from an existing system, it was expected that, given the same input, both systems would produce the same response. This is the case for the `GPS` and `IMU` sensor components. However, the modifications that were completed to the `HILInterface` in order to monitor and extract the emulates sensor values as they were exchanged between the Matlab HIL simultion and the existing flight control system resulted in the real-time requirements for the new system not being met. While the `GPS` and `IMU` component sensor telemetry was comparable with the existing system, `Estimator`, `Controller` and `Servo` component telemetry exhibited small but noticeable differences. The differences are as a result of algorithmic calculation not completing on time for some 20ms output periods, resulting in errors being accumulated when compared to the output of the original flight control system.

6.5 Flight Control System

6.5.1 Method

This section presents results collected while the new system hardware and software was tested in a HIL simulation. Four way-points were programmed into the system from the ground-station to form a lap, as described in section 6.4 and table 6.1. The results presented below document how the system performed.

As shown by the controller way-point telemetry in figure 6.32, two laps were flown before the simulation was stopped. (The next destination way-

Stellenbosch University https://scholar.sun.ac.za

point indicators in the subsequent graphs correlate with the way-points shown in figure 6.32.)

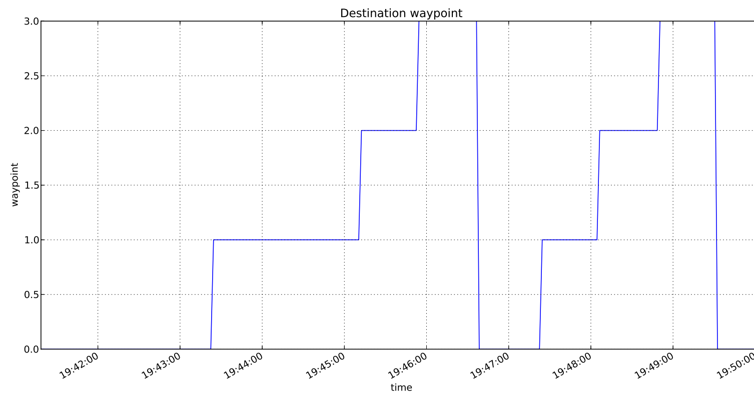


Figure 6.32: Next destination way-point selected by controller.

After manual take-off from the runway located at WP3, the autopilot was engaged in order to allow the aircraft to fly autonomously between the four way-points in a clock-wise manner starting at WP0. The GPS positional telemetry in figure 6.33 shows that the aircraft initially manoeuvred extensively around WP0, before proceeding to navigate between the remaining way-points. The reason for the manoeuvring was the autopilot algorithm working to attain the correct altitude of 160m after take-off, as can be seen in the IMU pitot-static pressure altitude telemetry shown in figure 6.34. The aircraft then correctly maintained the altitude after it was achieved.

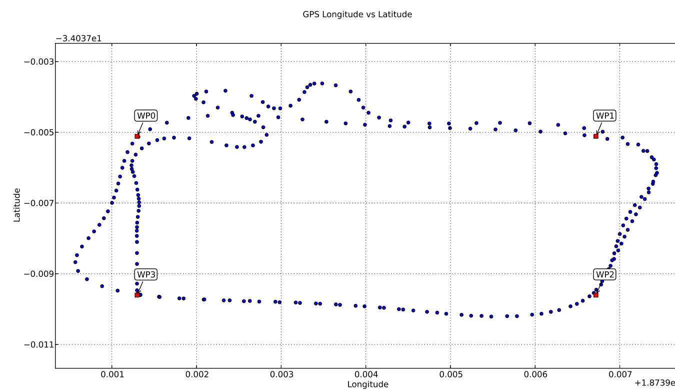


Figure 6.33: GPS Longitude vs Latitude.

Stellenbosch University https://scholar.sun.ac.za

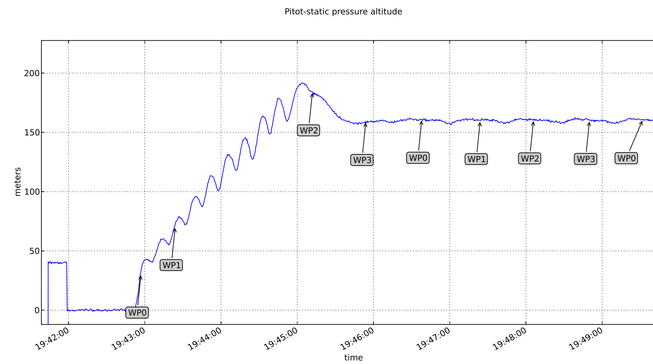


Figure 6.34: Pitot-static pressure altitude.

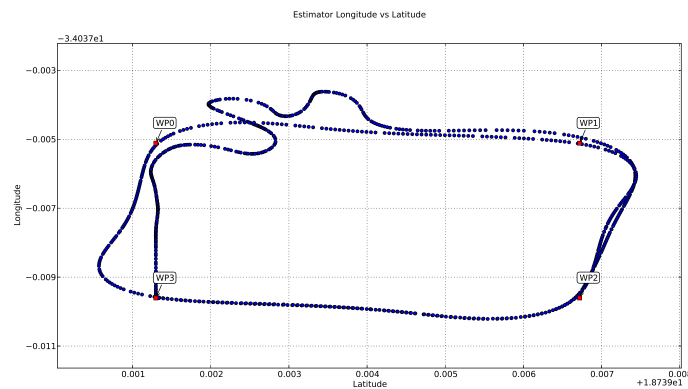


Figure 6.35: Estimator determined Longitude vs Latitude.

The position estimates calculated by the `Estimator` component, and shown in figure 6.35 correlated closely with the counterpart `GPS` measurements of figure 6.33. The pitot-static indicated airspeed `IMU` telemetry shown in figure 6.36 confirms that after the initial manoeuvring around WP0, the `Controller` regulated the aircraft airspeed at 22m/s.

Stellenbosch University https://scholar.sun.ac.za

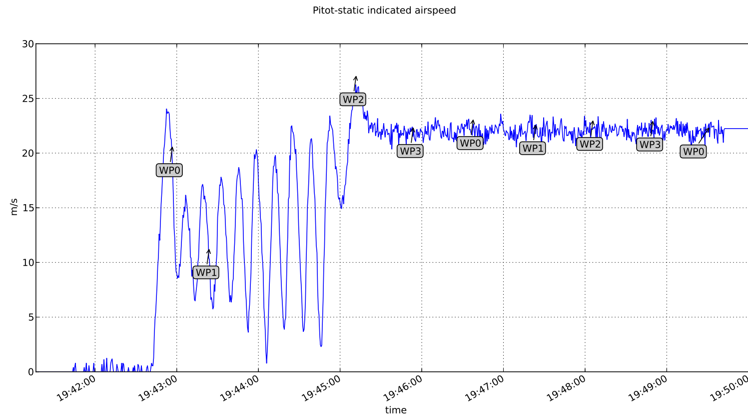


Figure 6.36: Pitot-static indicated airspeed.

In order to attain the correct altitude and airspeed, the aircraft had to initially throttle at full power, as seen in figure 6.37. Afterwards, only minor adjustments to the throttle power were required to maintain the correct airspeed. Similarly, the elevator also had to be engaged initially to pitch the aircraft up in order to reach the correct altitude. Once the altitude had been reached, the elevator actuator setting only required small changes to maintain a constant altitude, as seen in figure 6.38. The rudder and aileron were used to steer and roll the aircraft through the change in direction that occurred after the `Controller` selected the next destination way-point way-point, as can be seen in figures 6.39 and 6.40.

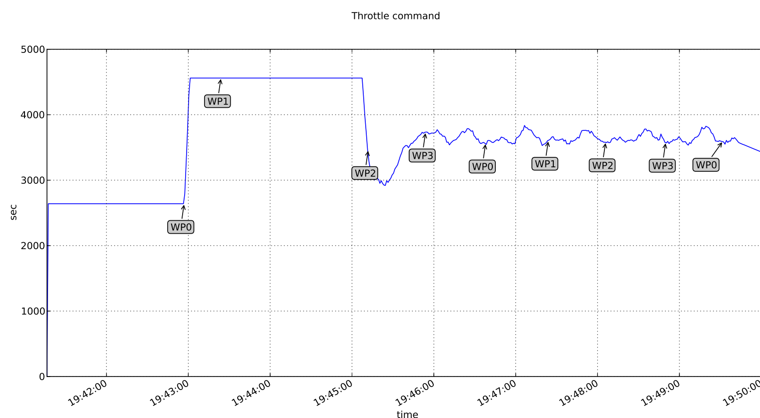


Figure 6.37: Actuator throttle command.

Stellenbosch University https://scholar.sun.ac.za

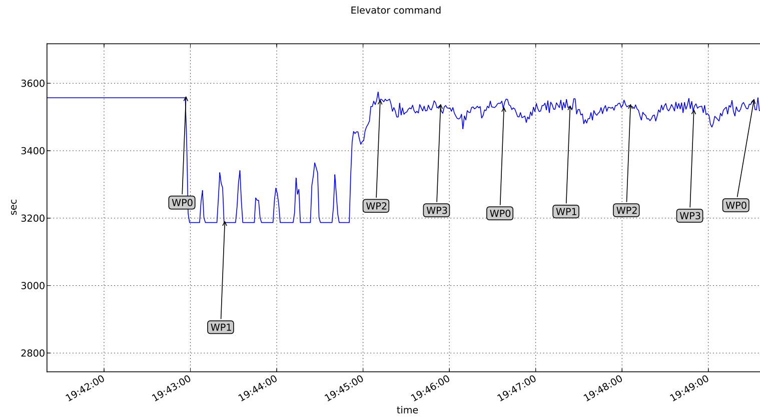


Figure 6.38: Actuator elevator command.

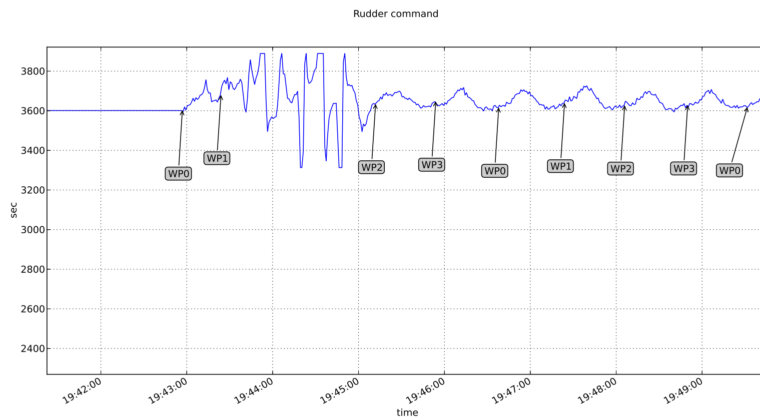


Figure 6.39: Actuator rudder command.

Stellenbosch University https://scholar.sun.ac.za

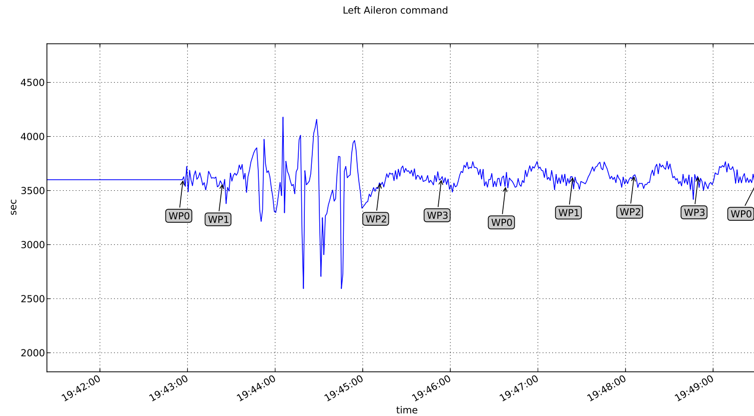


Figure 6.40: Actuator left aileron command.

6.5.2 Discussion

The telemetry showed that the aircraft successfully navigated through all the way-points in the right order during a HIL simulated two-lap flight. As figures 6.34 and 6.36 show, some initial manoeuvring by the autopilot was required to attain the configured altitude of 160m and airspeed of 22m/s. Once these had been achieved, the simulated aircraft proceeded as expected until the simulation was stopped.

Unlike the results from sections 6.4 and 6.6, figures 6.34, 6.36, 6.38 and others from this section show some form of oscillation occurring while the aircraft is manoeuvring towards the first way-point. This behaviours was observed during some simulation runs and is a consequence of the experimental setup described in section 6.4 and in particular due to the unavailability of an RC controller, which limited the ability to put the aircraft in close proximity of the initial way-point. Instead, the autopilot was engaged directly after take-off and the controller was responsible for achieving correct altitude, airspeed and position at the first way-point before proceeding to fly through the remaining way-points.

6.6 Distributed Flight Control System

6.6.1 Method

For this experiment, the flight control system was distributed over two target platforms: the majority of components were located on the original Gumstix SBC, while the `Estimator` was moved to a Linux-kernel based computer. Both hosts were connected via an Ethernet switch while a HIL simulation fed em-

Stellenbosch University https://scholar.sun.ac.za

ulated sensor values into the distributed flight control system. No software components had to be altered to achieve the distributed configuration: the instantiation of the `Estimator` component inside a run-time container was moved to the Linux-kernel based computer, while the rest of the configuration and launch sequence required to get the system executing remained constant. The telemetry results below document the performance of the distributed system while the configured way-points described in section 6.5.1 were flown in a HIL simulation.

The destination way-point telemetry shown in figure 6.41 indicate that the system completed two laps of the configured circuit before the simulation was stopped. (The next destination way-point indicators in the subsequent graphs correlate with the way-points shown in figure 6.41.)

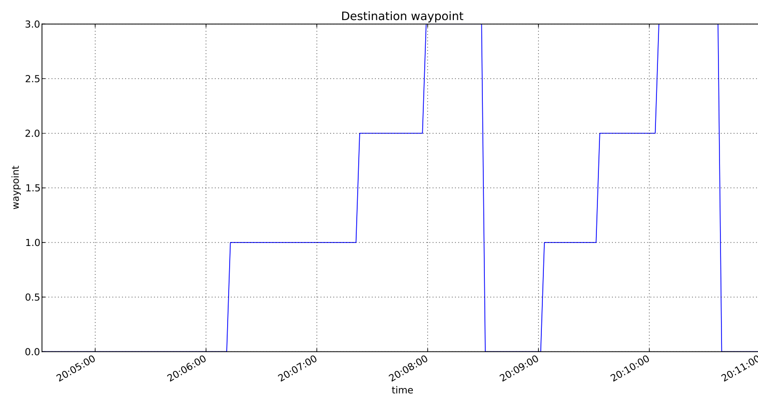


Figure 6.41: Waypoints for the distributed test.

As with the results presented in section 6.5.1, the aircraft took off manually from the runway located at WP3. After take-off, the autopilot was engaged and WP0 was selected as the next destination way-point. WP0 was quickly reached, at which point way-point WP1 was selected as the next destination way-point. However, as can be seen in figures 6.43 and 6.45, when the aircraft reached way-point WP0 it hadn't yet attained the correct altitude and airspeed, so manoeuvred until it achieved the correct values and then proceeded to fly to way-point WP1. Once WP1 was reached, WP2 was selected as the next destination way-point and finally WP3 before the lap was repeated. Figures 6.42 and 6.44 indicate the path followed by the aircraft as it flew through WP0 - WP3.

Stellenbosch University https://scholar.sun.ac.za

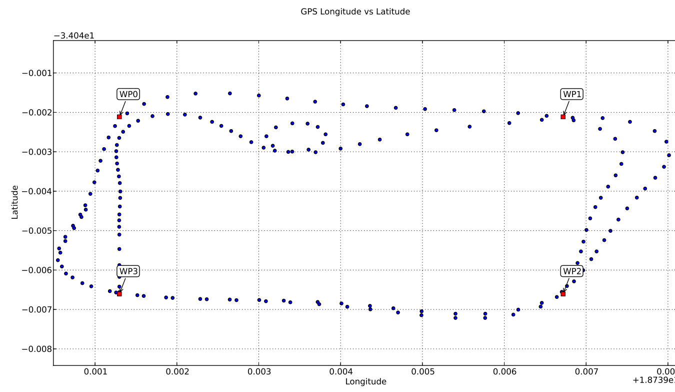


Figure 6.42: GPS Longitude vs Latitude.

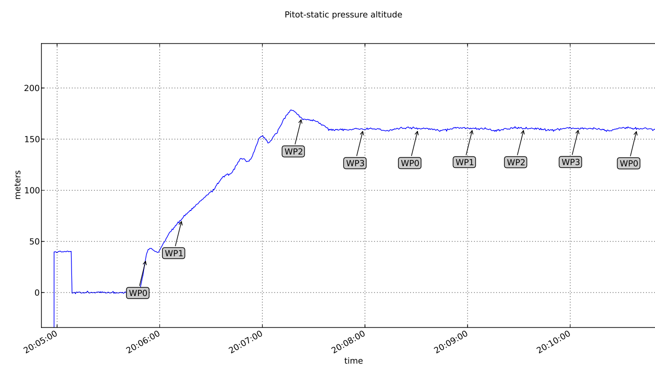


Figure 6.43: Pitot-static pressure altitude.

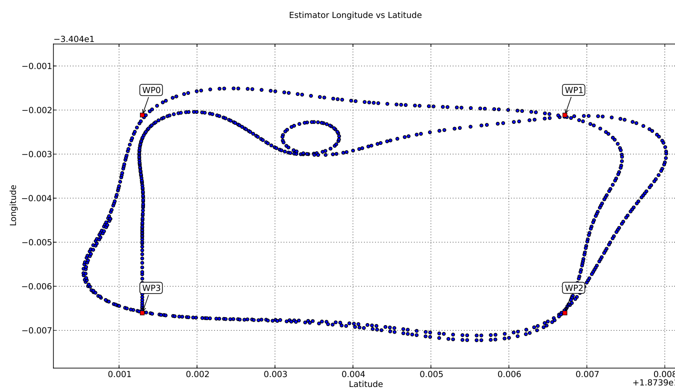


Figure 6.44: Estimator determined Longitude vs Latitude.

Stellenbosch University https://scholar.sun.ac.za

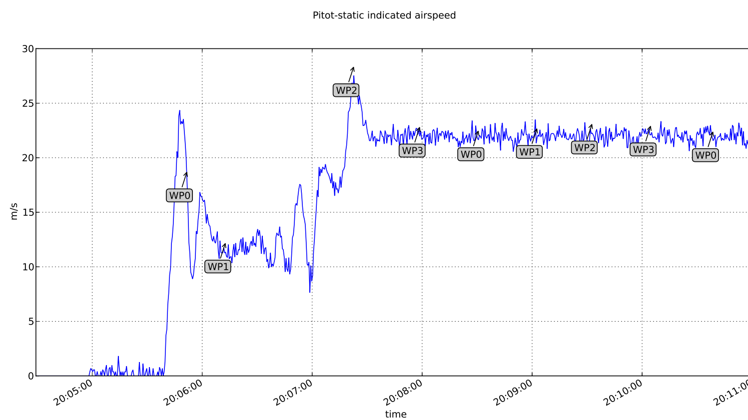


Figure 6.45: Pitot-static indicated airspeed.

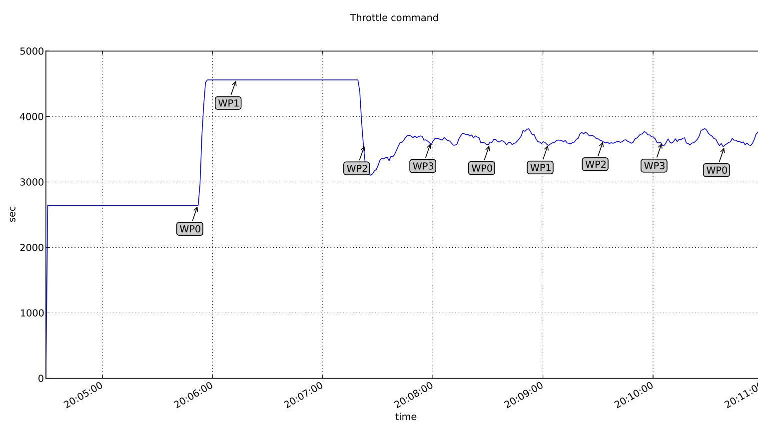


Figure 6.46: Actuator throttle command.

Stellenbosch University https://scholar.sun.ac.za

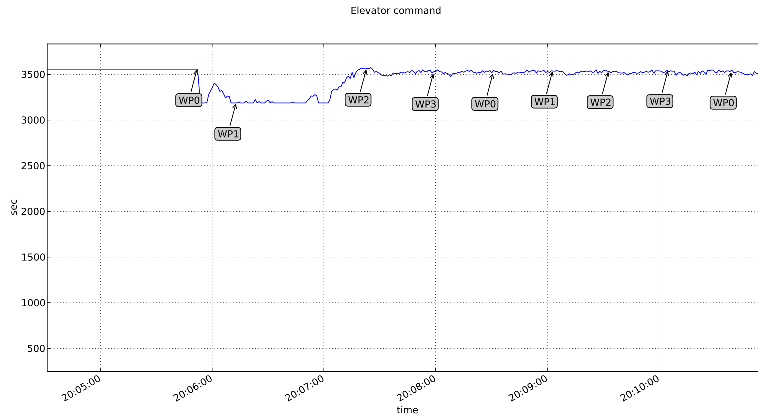


Figure 6.47: Actuator elevator command.

As was the case with the simulation described in subsection 6.5, the aircraft flew full-throttle initially to attain the correct airspeed (figure 6.46), while at the same time engaging the elevator to reach the desired altitude (figure 6.47). The rudder and aileron were used to steer and roll the aircraft through the change in direction that occurred once the current desired way-point had been reached and the next destination way-point had been selected, as is shown in figures 6.48 and 6.49.

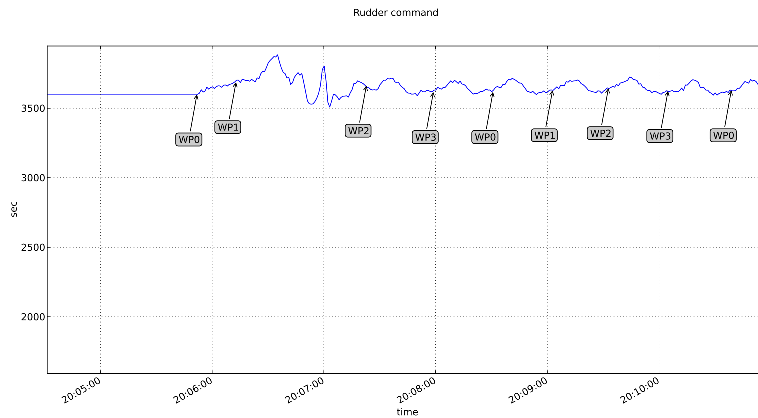


Figure 6.48: Actuator rudder command.

Stellenbosch University https://scholar.sun.ac.za

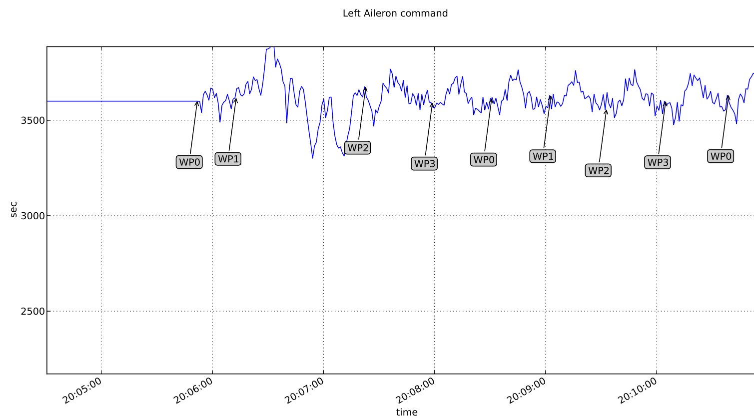


Figure 6.49: Actuator left aileron command.

6.6.2 Discussion

The telemetry results for the distributed flight control system showed that the aircraft performed as expected during the HIL simulation experiment. The aircraft reached and maintained the correct altitude and airspeed, while flying through the way-points in the correct order until the simulation was terminated.

6.7 Conclusion

The results from section 6.5 indicate that the developed avionics system was successfully able to maintain autonomous flight when tested in a HIL simulation environment. After some initial manoeuvring, the aircraft attained and maintained the correct altitude and airspeed, while flying through the programmed way-points in the correct order.

The parallel systems experiment described in section 6.4 showed that the new system performed in a similar manner to the existing system from which it was ported. Minor variations were observed in the actuator output, but these did not affect the ability of the system to fly autonomously. The source of the differences was traced back to real-time timing not being provided by the `HILInterface` component, which was modified to enable the experiment.

The distributed flight control system experiment described in section 6.6 indicated that components of the flight control application could be run over multiple processor boards while still successfully maintaining autonomous flight. This result indicated that it is possible for the system resources to be expanded when the need arises, without the need to re-develop existing hardware and

Stellenbosch University https://scholar.sun.ac.za

software. The result also validated the use of an Ethernet switch in a distributed system for real-time control applications.

The results of the HIL simulation experiments described in sections 6.5 and 6.6 as well as those described in section 6.2 indicated that the system was able to maintain real-time timing requirements.

The presented experimental results showed that the developed avionics system was able to maintain autonomous flight. The performance of the system was quantitatively similar to that of the existing system from which it was ported. The system was also able to operate in a real-time and distributed manner.

Stellenbosch University https://scholar.sun.ac.za

Chapter 7

Summary and Recommendations

7.1 Summary

In this project, options to address the limitations of the current avionics systems in use at the ESL were investigated. An architecture was designed to meet the objectives after identifying possible choices based on documented research, industry trends and open standards. The individual functions of the architecture were developed and integrated into a complete system, running on top of a real-time OS. The flight control algorithms were ported from an existing system in order to demonstrate the viability of the architecture and provide a basis for comparison. Experiments were conducted to determine whether the system met the stated objectives. Experimental results gathered from HIL simulations showed that the developed flight control system performed comparably to existing systems and was able to maintain autonomous flight. The system was also able to maintain autonomous flight when run in a distributed configuration. Provision was made for existing actuator technology to be re-used by developing a gateway that translated between CAN and Ethernet buses.

7.2 Recommendations

Since this project was verified using HIL simulations, a further development to advance the system would be for it to be packaged and industrialized into a form that is suitable for flight tests. Some additional developments are required for this: the CAN-Ethernet gateway was developed on an evaluation board - a replacement that can be used in flight tests needs to be identified. In the same sense, the Ethernet switch that was used for this project should also be replaced with an alternative that can be carried in one of the research vehicles - this could be as simple as removing the switch out of its housing to reduce the dimensions and weight of the switch.

Stellenbosch University https://scholar.sun.ac.za

Though CORBA Component based systems are actively used and developed in many telecommunications, medical, aerospace, defence and financial systems [56], its use is not as common outside industry and in research. Consequently, the volume of information related to component based development is much smaller compared to other libraries / middleware. As a result, the advantages afforded by component-based development, as used in this project to develop a DRE system, would have to be weighed up against the limitations of introducing a new programming paradigm in the ESL. Component-based middleware is currently being modernized to include support for the latest features introduced by "new" C++ standards, which could simplify the process of component-based development.

The future of component-based development appears set to evolve into the Unified Component Model (UCM), which is a new component model that aims to remove the underlying dependency on CORBA. The OMG has issued a request for proposal in order to start development on the UCM standard [57]. A stated purpose of the new model is for it to be "simple, lightweight, middleware-agnostic and flexible". This is a promising development, although it may be a while until a working implementation of the UCM standard is delivered, with no guarantee yet that it will be freely available.

Stellenbosch University https://scholar.sun.ac.za

Appendices

Stellenbosch University https://scholar.sun.ac.za

Appendix A

Software

A.1 Serial Communication Protocol

This section is reproduced from the description found in [10]. All serial communications were performed according to the string layout protocol, which was defined as follows:

```
$AA data * CS
```

where the identifiers can be described as follows:

- `$` indicates the start of a string.
- `AA` are two matching letters that uniquely identify the message.
- `data` is binary data of any length.
- `*` is the end of string delimiter.
- `CS` is a 1 byte checksum.

When constructing a message, the following rules must be adhered to:

- If any byte in the binary data string is a start or end of string delimiter, the respective data byte should be enqueued twice as to avoid ambiguity.
- The checksum is calculated by taking the exclusive OR (XOR) of all the bytes between the start and end of string delimiters, before any characters were doubled up as described by the previous rule.
- If the checksum is calculated to be an end of string delimiter character, the checksum is to be replaced with a '+' character as to avoid ambiguity.

Stellenbosch University https://scholar.sun.ac.za

List of References

[1] Lead, J.A.: Jsf avionics architecture definition appendices. *Arlington, USA: JAST Avionic Lead*, 1994.

[2] Schantz, R.E. and Schmidt, D.C.: Middleware for distributed systems: Evolving the common structure for network-centric applications. *Encyclopedia of Software Engineering*, vol. 1, 2002.

[3] Lea, D.: *Design patterns for avionics control systems.* Doug Lea., 1994.

[4] Gaum, R. and de Hart, R.: Gen micro obc firmware object documentation. Documentation delivered with v1.01 of ESL OBC firmware.

[5] Peddle, I.K.: *Autonomous flight of a model aircraft.* Master's thesis, Department of Electrical and Electronic Engineering, University of Stellenbosch, 2005.

[6] Roos, J.: *Autonomous Take-Off and Landing of an Unmanned Aerial Vehicle, University of Stellenbosch.* Master's thesis, Department of Electrical and Electronic Engineering, University of Stellenbosch, 2006.

[7] Carstens, N.: *Development of a Low-Cost Low-Weight Flight Control System for an Electrically Powered Model Helicopter.* Master's thesis, Department of Electrical and Electronic Engineering, University of Stellenbosch, 2005.

[8] Groenwald, S.: *Development of a Rotary-Wing Test Bed for Autonomous Flight.* Master's thesis, Department of Electrical and Electronic Engineering, University of Stellenbosch, 2005.

[9] Venter, J.: *Development of an experimental Tilt-Wing VTOL Unmanned Aerial Vehicle.* Master's thesis, Department of Electrical and Electronic Engineering, University of Stellenbosch, 2005.

[10] Hough, W.: *Autonomous Aerobatic Flight of a Fixed Wing Unmanned Aerial Vehicle.* Master's thesis, Department of Electrical and Electronic Engineering, University of Stellenbosch, 2007.

[11] Blaauw, D.: *Flight Control System for a Variable Stability UAV.* Master's thesis, Department of Electrical and Electronic Engineering, University of Stellenbosch, 2009.

Stellenbosch University https://scholar.sun.ac.za

[12] Gambier, A.: Real-time control systems: a tutorial. In: *Control Conference, 2004. 5th Asian*, vol. 2, pp. 1024–1031. IEEE, 2004.

[13] De Jager, A.: *The Design and Implementation of vision-based autonomous rotorcraft landing*. Master's thesis, Department of Electrical and Electronic Engineering, University of Stellenbosch, 2011.

[14] Integrated modular avionics wikipedia. `https://en.wikipedia.org/wiki/Integrated_modular_avionics`. Accessed: 2015-11-15.

[15] Jo, H.-C., Han, S., Lee, S.-H. and Jin, H.-W.: Implementing control and mission software of uav by exploiting open source software-based arinc 653. In: *Digital Avionics Systems Conference (DASC), 2012 IEEE/AIAA 31st*, pp. 8B2–1. IEEE, 2012.

[16] Eveleens, R.: Open systems integrated modular avionics - the real thing. Available at: `http://ftp.rta.nato.int/public/PubFullText/RTO/EN/RTO-EN-SCI-176/EN-SCI-176-04.pdf`, [12 October 2014], 2006. NATO.

[17] Tagawa, G.B. and e Souza, M.L.d.O.: An overview of the integrated modular avionics (ima) concept. *Proc. DINCON*, pp. 277–280, 2011.

[18] Kahn, A.: *The Design and Development of a Modular Avionics System*. Master's thesis, Georgia Institute of Technology, 2001.

[19] Ilarslan, M., Bayrakceken, M. and Arisoy, A.: Avionics system design of a mini vtol uav. In: *Digital Avionics Systems Conference (DASC), 2010 IEEE/AIAA 29th*, pp. 6–A. IEEE, 2010.

[20] Pastor, E., Lopez, J. and Royo, P.: Uav payload and mission control hardware/software architecture. *Aerospace and Electronic Systems Magazine, IEEE*, vol. 22, no. 6, pp. 3–8, 2007.

[21] Norris, R.B.: *A distributed flight control system architecture for small UAVs*. Master's thesis, Massachusetts Institute of Technology, 1998.

[22] Lee, K.C. and Lee, S.: Performance evaluation of switched ethernet for real-time industrial communications. *Computer standards & interfaces*, vol. 24, no. 5, pp. 411–423, 2002.

[23] Khazali, I., Boulais, M. and Cole, P.: Afdx software network stack implementation?practical lessons learned. In: *Digital Avionics Systems Conference, 2009. DASC'09. IEEE/AIAA 28th*, pp. 1–B. IEEE, 2009.

[24] Schmidt, D.C. and Vinoski, S.: The corba component model: Part1, evolving towards component middleware. *C/C++ Users Journal*, 2004.

[25] Sharp, D.C.: Reducing avionics software cost through component based product line development. In: *Digital Avionics Systems Conference, 1998. Proceedings., 17th DASC. The AIAA/IEEE/SAE*, vol. 2, pp. G32–1. IEEE, 1998.

Stellenbosch University https://scholar.sun.ac.za

[26] Paunicka, J.L., Mendel, B.R. and Corman, D.E.: The ocp-an open middleware solution for embedded systems. In: *American Control Conference, 2001. Proceedings of the 2001*, vol. 5, pp. 3445–3450. IEEE, 2001.

[27] Yussof, H., Capi, G., Nasu, Y., Yamano, M. and Ohka, M.: A corba-based control architecture for real-time teleoperation tasks in a developmental humanoid robot. *International Journal of Advanced Robotic Systems*, vol. 8, no. 2, pp. 29–48, 2011.

[28] Doherty, P., Haslum, P., Heintz, F., Merz, T., Nyblom, P., Persson, T. and Wingman, B.: A distributed architecture for autonomous unmanned aerial vehicle experimentation. In: *Distributed Autonomous Robotic Systems 6*, pp. 233–242. Springer, 2007.

[29] Schmidt, D.C. and Kuhns, F.: An overview of the real-time corba specification. *Computer*, vol. 33, no. 6, pp. 56–63, 2000.

[30] Deng, G., Gill, C., Schmidt, D.C. and Wang, N.: Qos-enabled component middleware for distributed real-time and embedded systems. *Handbook of Real-Time And Embedded Systems*, pp. 15–1, 2007.

[31] Otte, W.R., Gokhale, A., Schmidt, D.C. and Willemsen, J.: Infrastructure for component-based dds application development. In: *ACM SIGPLAN Notices*, vol. 47, pp. 53–62. ACM, 2011.

[32] Harrison, T.H., Levine, D.L. and Schmidt, D.C.: The design and performance of a real-time corba event service. *ACM SIGPLAN Notices*, vol. 32, no. 10, pp. 184–200, 1997.

[33] Pedersen, R.: Object request broker software technology: applications in an advanced open systems avionics architecture. In: *Digital Avionics Systems Conference, 1997. 16th DASC., AIAA/IEEE*, vol. 1, pp. 5–2. IEEE, 1997.

[34] Trombetti, G., Gokhale, A., Schmidt, D.C., Greenwald, J., Hatcliff, J., Jung, G. and Singh, G.: An integrated model-driven development environment for composing and validating distributed real-time and embedded systems. In: *Model-driven Software Development*, pp. 329–361. Springer, 2005.

[35] Sharp, D.C.: Object-oriented real-time computing for reusable avionics software. In: *Object-Oriented Real-Time Distributed Computing, 2001. ISORC-2001. Proceedings. Fourth IEEE International Symposium on*, pp. 185–192. IEEE, 2001.

[36] OMG: *Lightweight CORBA Component Model RFP, realtime/02-11-27 edition.* Object Management Group, 2002.

[37] Karsai, A., Kereskenyi, R. and Mahadevan, N.: A real-time component framework: Experience with ccm and arinc 653. In: *IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*. 2010.

Stellenbosch University https://scholar.sun.ac.za

[38] Dubey, A., Emfinger, W., Gokhale, A., Karsai, G., Otte, W.R., Parsons, J., Szabó, C., Coglio, A., Smith, E. and Bose, P.: A software platform for fractionated spacecraft. In: *Aerospace Conference, 2012 IEEE*, pp. 1–20. IEEE, 2012.

[39] Object management group, model driven architecture. `http://www.omg.org/mda/specs.htm`, . Accessed: 2014-11-15.

[40] Generic modelling environment. `http://w3.isis.vanderbilt.edu/Projects/gme/`. Accessed: 2014-11-15.

[41] Linux RT-PREEMPT frequently asked questions. `https://rt.wiki.kernel.org/index.php/Frequently_Asked_Questions`. Accessed: 2014-11-15.

[42] National Institute of Standards and Technology introduction to linux for real-time control. `http://www.aeolean.com/html/RealTimeLinux/RealTimeLinuxReport-2.0.0.pdf`. Accessed: 2014-11-20.

[43] Krasner, G. and Pope, S.: A cookbook for uisng the model view controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming*, 1988.

[44] Gill, C.D., Levine, D.L. and Schmidt, D.C.: The design and performance of a real-time corba scheduling service. In: *Challenges in Design and Implementation of Middlewares for Real-Time Systems*, pp. 3–40. Springer, 2001.

[45] Sharp, D.C.: Avionics product line software architecture flow policies. In: *Digital Avionics Systems Conference, 1999. Proceedings. 18th*, vol. 2, pp. 9–C. IEEE, 1999.

[46] Doerr, B.S. and Sharp, D.C.: Freeing product line architectures from execution dependencies. In: *Software Product Lines*, pp. 313–329. Springer, 2000.

[47] Gill, C.D., Cytron, R.K. and Schmidt, D.C.: Multiparadigm scheduling for distributed real-time embedded computing. *Proceedings of the IEEE*, vol. 91, no. 1, pp. 183–197, 2003.

[48] Paunicka, J.L., Mendel, B.R. and Corman, D.E.: Open control platform: A software platform supporting advances in uav control technology. *Software-Enabled Control: Information Technology for Dynamical Systems*, pp. 39–62, 2005.

[49] Object management group, deployment and configuration of component-based distributed applications. `http://www.omg.org/spec/DEPL/`, . Accessed: 2014-11-15.

[50] Deng, G., Balasubramanian, J., Otte, W., Schmidt, D.C. and Gokhale, A.: Dance: A qos-enabled component deployment and configuration engine. In: *Component Deployment*, pp. 67–82. Springer, 2005.

[51] Yocto project. `https://www.yoctoproject.org/about`. Accessed: 2014-11-20.

Stellenbosch University https://scholar.sun.ac.za

[52] Schmidt, D.C.: Reactor: An object behavioral pattern for concurrent event demultiplexing and dispatching. 1995.

[53] Schmidt, D.C. and Pyarali, I.: The design and use of the ace reactor. *URL http://www. cs. wustl. edu/˜ schmidt/PDF/reactor-rules. pdf.*

[54] Gaum, D.R.: *Agressive flight control techniques for a fixed wing unmanned aerial vehicle.* Master's thesis, 2009.

[55] Schmidt, A.G.D.C. and Gray, N.W.: Model driven middleware. *Middleware for Communications*, p. 163, 2004.

[56] Tao users. `http://www.dre.vanderbilt.edu/~schmidt/TAO-users.html`. Accessed: 2014-11-20.

[57] Unified component model rfp. `http://www.omg.org/cgi-bin/doc?mars/13-09-10`. Accessed: 2014-11-20.