Verifying the I/O-System for the new Aircraft Simulator at Saab

Examensarbete utfört i elektroteknik vid Saab Tannefors av Jesper Erlingborn

LiTH-ISY-EX-ET--15/0448--SE

Linköping 2015
Verifying the I/O-System for the new Aircraft Simulator at Saab

Examensarbete utfört i elektroteknik
vid Saab Tannefors
av

Jesper Erlingborn

LiTH-ISY-EX-ET--15/0448--SE

Handledare:  Elisabet Ressner
Saab Aeronautics

Lars Karlsson
Saab Aeronautics

Per Eriksson
Saab Aeronautics

Per Nicolaisen
Saab Aeronautics

Arta Alvandpour
Linköping University

Examinator:  Dr. J Jacob Wikner
Dept. of Electrical Engineering
Linköping University

Linköping  -- August 2015
Språk
Language
☐ Svenska/Swedish
☒ Engelska/English

Typ av publikation
Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Rapport
☐ Annat (ange nedan)

ISSN

ISBN

ISRN
LiTH-ISY-EX-ET--15/xxxx—SE

Serietitel och Serienummer/ISSN
Titles of series, numbering

Presentation datum

Publiceringsdatum

Institution och avdelning
Institutionen för Systemteknik
Department of Electrical Engineering

URL för elektronisk version

Titel
Title
Verifying the I/O-System for the new aircraft Simulator at Saab

Författare
Author
Jesper Erlingborn’

Sammanfattning
Abstract

This thesis revolves around how to verify that the new aircraft simulator that is currently under development at Saab Aeronautics functions as intended. Because the system is still under development, the assignment is about developing various suggestions on how to verify that the new simulator is working properly. After various test proposals have been developed for the system, I will put them to the test with the help of National Instruments software. I will also make a world study to compare how other companies develop their simulators.

The development of a new simulator is not an easy task, new technical systems must be developed which will result in various problems to be solved. Systems must also be regularly tested to verify that everything works as it should. Flight simulators are a common and useful tool for control system design as well as verification and validation, and are used extensively throughout the aviation industry to evaluate system performance.

Nyckelord
Keywords    Simulator, Input, Output, I/O-System, Saab, aircraft
Abstract

This thesis revolves around how to verify that the new aircraft simulator that is currently under development at Saab Aeronautics functions as intended. Because the system is still under development, the assignment is about developing various suggestions on how to verify that the new simulator is working properly. After various test proposals have been developed for the system, I will put them to the test with the help of National Instruments software. I will also make a world study to compare how other companies develop their simulators.

The development of a new simulator is not an easy task, new technical systems must be developed which will result in various problems to be solved. Systems must also be regularly tested to verify that everything works as it should. Flight simulators are a common and useful tool for control system design as well as verification and validation, and are used extensively throughout the aviation industry to evaluate system performance.
Acknowledgments

First of all I’d like to thank Per Eriksson, my supervisor at Saab Aeronautics, for all the help you’ve given me with the I/O-System and LabVIEW. I’m also grateful for the pleasant work environment created by you and your colleagues at Saab.

Secondly, I would like to thank Bjorn Beckman from National Instruments that made it possible for me to fulfil this thesis through the lending of hardware and software.

Last, I want to thank Jacob Wikner, my examiner at the university, for your positive attitude in general and for your help in this thesis, as well as in previous courses I’ve had with you.

Linköping, September 2015
Jesper Erlingborn
# Contents

1 INTRODUCTION ................................................................. 15  
1.1 Background ..................................................................... 15  
1.2 Motivation ...................................................................... 15  
1.3 Purpose with this Thesis ................................................. 15  
1.4 Questions for the thesis ..................................................... 16  
1.5 Constraints ................................................................... 16  
1.6 Method ........................................................................ 16  
2 BACKGROUND .................................................................... 19  
2.1 Brief Introduction to Flight Simulators ................................. 19  
2.2 Soft Simulators ................................................................. 21  
2.3 System Rigs ................................................................... 22  
2.4 UDP VS TCP ................................................................ 24  
2.5 Introduction to CompactRIO and LabVIEW ....................... 25  
2.6 Components Used .......................................................... 27  
2.6.1 NI crIO-9022 ......................................................... 27  
2.6.2 NI-9403 .................................................................. 28  
2.6.3 NI-9263 .................................................................. 29  
2.6.4 NI-9215 .................................................................. 30  
2.6.5 NI-9871 .................................................................. 31  
3 THEORY ........................................................................... 33  
3.1 Development of a Novel Low-Cost Flight Simulator ............ 33  
3.1.1 Introduction .............................................................. 33  
3.1.2 Aircraft Dynamics Mathematical Model ......................... 34  
3.1.3 Motion System .......................................................... 35  
3.1.4 Semi-Virtual Cockpit System ....................................... 36  
3.1.5 Control Loading System ............................................. 37  
3.1.6 Other Subsystems ....................................................... 37  
3.2 Development of a Pilot-in-the-loop Flight Simulator using NASA’s Transport Class Model ........................................... 37  
3.2.1 Introduction .............................................................. 37  
3.2.2 TCM Simulation Model .............................................. 38  
3.2.3 Simulator Hardware .................................................. 39  
3.3 Overall System Description at Saab ................................... 41  
3.3.1 System Description .................................................... 41  
3.3.2 CompactRIO .......................................................... 42  
3.4 Design Decisions ............................................................. 44  
3.4.1 Main Timing ............................................................. 44  
3.4.2 Supported Signals ...................................................... 44  
4 METHOD ........................................................................ 47  
4.1 Implementation and Experimentation ................................... 47  
4.1.1 Brief Introduction of the various test cases ...................... 47  
4.1.2 Setting up a new project in LabVIEW ............................ 48  
4.1.3 Complete NI I/O-System Overview ............................... 54
### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>A/C</td>
<td>Aircraft</td>
</tr>
<tr>
<td>ACSIM</td>
<td>Aircraft Simulator</td>
</tr>
<tr>
<td>A/C-SIMULATOR</td>
<td>Aircraft Simulator</td>
</tr>
<tr>
<td>ADC</td>
<td>Analog-to-Digital Converter</td>
</tr>
<tr>
<td>AI/AO</td>
<td>Analog Input / Analog Output</td>
</tr>
<tr>
<td>ARINC</td>
<td>Aeronautical Radio INC</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial Off The Shelf</td>
</tr>
<tr>
<td>cRIO</td>
<td>Compact Reconfigurable I/O</td>
</tr>
<tr>
<td>DOF</td>
<td>Dimensions Of Freedom</td>
</tr>
<tr>
<td>DI/DO</td>
<td>Digital Input / Digital Output</td>
</tr>
<tr>
<td>EUT</td>
<td>Equipment Under Test</td>
</tr>
<tr>
<td>FAA</td>
<td>Federal Aviation Administration</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Array</td>
</tr>
<tr>
<td>GTM</td>
<td>Generic Transport Model</td>
</tr>
<tr>
<td>HIL</td>
<td>Hardware In the Loop</td>
</tr>
<tr>
<td>H/W</td>
<td>Hardware</td>
</tr>
<tr>
<td>ISL</td>
<td>Illinois Simulator Laboratory</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>IOS</td>
<td>Introduction and Operator Station</td>
</tr>
<tr>
<td>LoC</td>
<td>Loss of Control</td>
</tr>
<tr>
<td>NI</td>
<td>National Instruments</td>
</tr>
<tr>
<td>PITL</td>
<td>Pilot In the Loop</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real-Time Operating System</td>
</tr>
<tr>
<td>RMS</td>
<td>Root Mean Square</td>
</tr>
<tr>
<td>SIL</td>
<td>Software In the Loop</td>
</tr>
<tr>
<td>S/W</td>
<td>Software</td>
</tr>
<tr>
<td>SAR</td>
<td>Successive Approximation Register</td>
</tr>
<tr>
<td>SUT</td>
<td>Subject Under Test</td>
</tr>
<tr>
<td>TES</td>
<td>Tactical Environment</td>
</tr>
<tr>
<td>TCM</td>
<td>Transport Class Model</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>VR</td>
<td>Virtual Reality</td>
</tr>
<tr>
<td>V&amp;V</td>
<td>Verification and Validation</td>
</tr>
<tr>
<td>VIZ</td>
<td>Visual Environment</td>
</tr>
<tr>
<td>VI</td>
<td>Virtual Instrument</td>
</tr>
</tbody>
</table>
1 Introduction

This chapter will give you an introduction to the work in this thesis together with the problem formulation.

1.1 Background

Simulation is the imitation of something real that is available or not yet available. If the system that is being simulated is an aircraft, then it is a flight simulator.

A Flight simulator is a computer system with software that replicates the environment and systems that allows the user to experience what it feels like to pilot a specific aircraft.

The Flight simulators are used extensively throughout the aviation industry to evaluate system performance and a great tool for system design, verification and validation of the system. Verification and validation, used to verify that the system is working in the intended manner is a crucial step in any development of a safety-critical system.

1.2 Motivation

This thesis will focus on the development of various tests for the new flight simulator that is currently being developed at Saab. The simulator is divided into different components hierarchically. For simplicity, the real aircraft hardware is located in the lowest region of the hierarchy and in the top sits the simulation computers executing models and I/O in real time. In between there is an I/O-System and some devices that manages the adaptation of the different signals that go in and out from the I/O-System. The I/O-System is used to manage the communication between computers and the simulated aircraft hardware.

The problem that arises is because the simulator is still under development and the new I/O-System is currently under verification and validation to make sure that the format used to talk to the I/O-System is consistent with the code used for the simulation computers.

My work will therefore involve learning the I/O-System thoroughly at low level and perform tests on the system to ensure that the signals are correct and arrive in the expected way. After I have come up with various test proposals for how to test the system, I will put them to the test with the help of software and hardware that I borrowed out of National Instruments.

1.3 Purpose with this Thesis

The aim of this thesis is to facilitate some elements in the development of the new flight simulator. At the moment, they are not sure whether the format that is used to talk to the I/O-System is consistent with the actual code that is running on the simulation computers in the system. What is missing in the development is to come up with good ideas on how to verify the new system.
If the thesis is carried out, it would mean that the development team can be sure on how the signals go. This will result in that they can take the design down to the real aircraft hardware and be sure that it works. The I/O System cannot be used against the real aircraft hardware until it has been tested and verified that it really works, otherwise, in worst case scenario damage expensive hardware.

So if problems arise when testing for real, they can at least be sure that the fault lies elsewhere if the thesis is carried out. This means that the development time to the real aircraft hardware test will decrease radically, which in turn means less development cost for the company.

1.4 Questions for the thesis

This thesis will investigate

1. How should we proceed in order to ensure that the data transfer inside the input/output system is working correctly?

2. How should we do to test so the format that talks with the I/O System is consistent with the real code running on the simulation computers?

1.5 Constraints

The flight simulator consists of a variety of smaller subsystems which together interact with each other to form a functioning system. In the final system there are subsystems with both software and hardware, together connected to a functioning system.

This thesis will only focus on the I/O System which is hardware with an integrated FPGA used to be programmed by the user, which means both software and hardware interact.

If you look at the work from a social and ethical perspective, it is actually a subsystem for an aircraft that is supposed to be used to protect the country and create a safer place. But the question is whether it really becomes a safer place by building a high-tech aircraft with high-tech weapons on it. The entire project may instead end up with that other countries see Sweden as a threat and thus it may become an unsafe place. People also have different views on such a large industrial projects costing Sweden enormous sums of money.

1.6 Method

This thesis has been divided into three different parts. The first part, I have chosen to call the theory part, it has been about gathering as much information about the system as possible, so I know what I have to work with as well as the difficulties in the system. Which meant reading about CompactRIO and everything associated with the hardware to do. It also meant learning LabVIEW at a higher level when there were no major knowledge in the software before the thesis started.

Also performed a great surrounding world study to learn and get a broader knowledge and perspective on how other companies have made to develop their aircraft simulators.
After I had borrowed all the hardware and software from National Instruments, I began to
draw up the I/O-System using LabVIEW to have a program that I could later perform tests on.
The second part, I have chosen to call for the proposal stage. This part of the work has been to
develop proposals on how to test the I/O-System so that it meets their needs as described
earlier in the report.
The last part, I have chosen to call for test and verification step. This step is to try the test
methodology I took up in step two. I do this by re-creating ideas in LabVIEW software to run
the actual tests on the I/O-System.
2 Background

2.1 Brief Introduction to Flight Simulators

A flight simulator can broadly be described as some sort of cabin with a throttle and a stick which the operator use to control the aircraft. The stick is then connected to a simulation computer, where it is based on what the pilot does with the lever that will decide how the aircraft will behave.

The modules that calculate this are models of the aircrafts aerodynamics and flight mechanical properties. This information is transmitted further to an image generator, generating graphics for the environment. The image is presented to the pilot by means of some sort of display. The operator is also notified of their instruments [9].

The simulators in use today are much more advanced than what has been described above. In order to simulate the functions that are not represented by hardware in a simulator, powerful simulation computers are used.

A visual environment is used to represent the surroundings and there is also a tactical environment that makes it possible to simulate friendly and enemy aircraft, ships and other vehicles during exercises.

A development simulator is used most extensively to develop and verify software in the aircraft computers. Also used for verification and validation of the aircraft's hardware. A development simulator can be regarded as a test instrument to test an item in the aircraft to verify that it works, SUT (Subject Under Test).

There are two different types of development simulators and they are system Rigs and Soft simulators [8].
The picture above shows a generalized view of how the structure looks like in a flight simulator with all its components here at Saab. The first component is called Surrounding Environment and this will display all the graphics for the user who operates the simulator, such as weather, radar, tactical environment and weapon.

The next step as you can see in the image above is named aircraft systems which is modelled aircraft hardware such as the aircraft engine, brakes, landing gear and dynamic forces. The External Stores will include the plane's external devices such as different weapons and sensors [8].

The next step is called physic2Logic and this is the I/O-System which in turn means that it will handle all communication between the hardware and the software. Note that this is only for the system rig that I will go through later. My work here at the simulator department at Saab will be to study this component closer and identifying ways to validate and verify. Then we have all the ECUs, which means real aircraft hardware with real software such as hydraulic system and fuel systems [8].

One of the blocks above is called Tactical systems and it manages the aircrafts tactical characteristics such as radar, sensors, and electronic warfare.

Last in the chain, we have all the safety-critical systems and mission computers that all previous systems are connected to in order to send information to the pilot which will show up on the plane's HUD (Head Up Display) and HDD (Head Down Display). The information can be anything from how much fuel that remains to enemy aircrafts [8].
2.2 Soft Simulators

Soft simulators contain no real hardware at all so everything is modelled. Both simulators follow the simple "principle", which means putting the aircraft unit in an apparatus bed and trick it to believe that it is in a real aircraft. For simplicity this means stimulating the units I/O with real signals [8].

![Soft Simulator Architecture](image)

*Figure 2.1 Soft Simulator Architecture*

The picture above show a simple system diagram of how a soft simulator looks like with all containing parts. The picture above is a bit misleading as it says I/O in the image. There is no real hardware in a soft simulator so the picture show you software I/O.
2.3 System Rigs

The system Rig contain as much as possible of the real aircraft equipment. What distinguishes the simulators apart are the devices currently in use. Other devices and systems are modeled [8].

System Rigs at Saab is built with a real aircraft as a base.

Such equipment which will not in a simulator, such as aircraft engine, are removed and replaced with different models.

An interface with the remaining devices are made and a model that simulates the outside world are developed. The reason that the simulators are built in this way is that as far as possible be able to test flight systems without having to fly for ”real”. Some benefits of simulators is that they are on the ground, which means that in a simple way measure the buses and devices while the aircraft ”flies”.

It is also possible to perform high-risk samples without the chance of wrecking a real aircraft. Of course there are also cost benefits of flying in a simulator instead of flying for real. After a system has been tested in a simulator, flight test must be performed, but the risks reduces significant if the system is first tested in a simulator [9].
The picture above show a simple system diagram of how a system Rig looks like with all containing parts. Below is a brief description for each of the containing blocks in the picture above.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOS</td>
<td>Instruction and Operator Station. Includes features to configure, control and monitor the simulation</td>
</tr>
<tr>
<td>Simulation Computers</td>
<td>Powerful computers that executes models and I/O in real time</td>
</tr>
<tr>
<td>TES</td>
<td>Tactical Environment, used to simulate friendly and enemy aircraft</td>
</tr>
<tr>
<td>VIZ</td>
<td>Visual Environment</td>
</tr>
<tr>
<td>I/O</td>
<td>Hardware, used for communication between the simulation computers and aircraft hardware</td>
</tr>
<tr>
<td>HW ANP</td>
<td>Adaption device that scales the signal</td>
</tr>
<tr>
<td>SUT</td>
<td>The hardware that is currently being tested (Subject Under Test)</td>
</tr>
</tbody>
</table>
The system Rig has multiple different purposes, making it versatile. Common uses are Software Development, Verification, Pilot training, Model development and Support simulation.

Below are various examples of applications and the requirements imposed on the simulators.

**Software Development**: The system developers test and develop their functions and code in the type real devices. This requires accurate models and good simulation and registration opportunities [10].

**Verification**: Verifying tests is done before the features is introduced to the aircraft. Demands like this use, requires that the simulators will occur real type and the simulator configuration is verified. There are also requirements on the Test Worthiness and registration opportunities [10].

**Pilot Training**: The pilots are training on the aircraft system before the flight, which requires that the outside world is good, that the simulators occurs type real and the cabin looks like it does in the real aircraft. It must also be possible to use error simulation [10].

**Model development**: Model development means that the simulators are used for the development of new and improvement of existing models. This application requires that the simulators have a real type behavior, that it is possible to record what is happening and that it is possible to measure whether the models executes at the time it is expected that they will execute [10].

**Support simulation**: This means that the entire test organization is trained before a flight test. The requirements for real type connection to measuring systems for flight test and it must be possible to carry out error simulations [10].

In conclusion, it must be possible to simulate the environment to the onboard computers in real time as well as it gets. The simulators must also be able to manage all versions and editions of the real type system. Other requirements for the simulators are that it must be possible to download the real type computers with software, communicate with other tactical systems and visualize the world as well as possible. Last but not least, the simulators must be user friendly [9].

### 2.4 UDP VS TCP

The Internet Protocol suite is the computer networking model and set of communications protocol which used on the Internet and on the Computer networks. It is commonly known as TCP/IP because it is the most important protocols. The TCP/IP (Transmission Control Protocol / Internet protocol) is the most basic protocol of the internet and were the first networking protocols defined in the standard.

TCP/IP provides point-to-point connectivity specifying how data is packaged, addressed, transmitted, routed and received at the destination. Point-to-point communication means that each communication is from one point (or host computer) in the network to another point or host computer.
The TCP/IP model is organized into four different layers. So from the lowest to highest, the layers are Link layer, Internet layer, Transport layer and Application layer [11]. One of the basic protocols of TCP/IP is called User Datagram Protocol (UDP) which is used in the aircraft Simulator at Saab to handle the communication between the various components.

Like the Transmission Control Protocol, UDP uses the Internet Protocol to actually get a data unit called datagram, from one computer to another. The difference between UDP and TCP is that UDP does not provide the service of dividing the messages into packets (datagrams) and reassembling it at the other end. Specifically, UDP does not provide sequencing of the order that the packets and data arrives in. This means that the program that is using UDP must do controls to check that the entire message has arrived and is in the right order.

To save processing time over network applications, the User Datagram Protocol is preferred over Transmission Control Protocol [11].

2.5 Introduction to CompactRIO and LabVIEW

The new flight simulator at Saab will receive a brand new developed I/O-System. The old I/O-System that was in the previous simulators were in house developed by Saab.

The old I/O-System consists of different cards that manages the data transfers in different ways, these are adapter cards, DI/DO, AI/AO and Serial communication.

What is special about the current cards in use is that each card has its own built-in FPGA that can be programmed.

The new I/O-System, currently has no name, will be developed in a completely different way. This system will be based on technology from suppliers and will therefore not be built by Saab itself. The supplier is National Instruments and the product in use for this purpose is called CompactRIO.

The difference between the old I/O-System and the new CompactRIO system is that the new system only has one built-in FPGA to control all units.

CompactRIO is the name of the system and all its components which together form a working system. CompactRIO consists of a processor running a real-time operating system (RTOS), a FPGA that is reconfigurable and I/O-Modules that is interchangeable. The I/O-Modules in the CompactRIO system will have the same features as the old cards in the old I/O-System, except that the old cards had its own built-in logic on each card.
The logic for the FPGA in CompactRIO is programmed with the help of LabVIEW software from National Instruments.

LabVIEW is a graphical programming language used to develop sophisticated control systems using graphical icons and wires that resemble a flowchart. The use of LabVIEW software is the best way to create user-defined programs more quickly because it offers integration with thousands of hardware devices and provides built-in libraries for advanced controls, analysis and data visualization [12].

The CompactRIO system consists of three hardware components and these are, Real-Time Controller, I/O-FPGA Chassis and I/O-Modules of your own choice.

The Real-Time controller has a built-in processor used to execute LabVIEW Real-Time applications, offering execution tracing, on board data logging and communication with other units [12].

The reconfigurable I/O-FPGA chassis is the most central part of the system architecture. What makes this device so special is that the I/O-Modules are directly connected to the FPGA, which allows the system to be lightning fast. No buses are needed when the modules are directly connected, which means that the user does not notice any major delays in the system response. Without having to program the FPGA itself, it is in default mode set to communicate with the I/O-Modules and send the information to the Real-Time processor. If you want to further develop the system, you can reprogram the FPGA until you are satisfied [12].

The I/O-Modules contain signal conditioning, conversion circuitry, isolation and built-in connectivity for direct connection to industrial sensors/actuators. By offering a variety of connection options for the I/O-Modules and even integration of the connection boxes in the modules, means that the CompactRIO system will require less space which in turn means that the system will be more compact and manageable [12].
2.6 Components Used

In the following text I will describe the components that I have used in the thesis.

2.6.1 NI cRIO-9022

The NI-cRIO-9022 embedded controller features an industrial 533 MHz Freescale MPC8347 real-time processor for deterministic and reliable real-time applications. This particular embedded controller is designed for extreme ruggedness, reliability and low power consumption with the help of dual 9 to 35 VDC supply inputs that deliver isolated power to the compactRIO chassis. The operating temperature range is between -20 to 55 °C. The best thing with the CompactRIO 9022 chassis is that it accepts 6 to 35 VDC power supply input during operation, which means that it can run in remote applications for long periods using only batteries or solar power [13].

![Image](image-url)

Figure 2.4 NI cRIO-9022

In this project there are three modules attached to the cRIO-9022: the NI-9401 digital input-output module, the NI-9263 analogue output module, and the NI-9215 analogue input module. The last module called NI-9871 will not be used in this thesis but later on in the flight simulator development.
The NI-9403 is a digital input and output module for CompactRIO and CompactDAQ chassis. In total it has 32-channels and the user have a lot of different ways to configure it. You can configure the direction of each digital line on the NI 9403 for input or output. For CompactRIO, you can use the LabVIEW FPGA Module to program the NI-9403 for implementing custom, high-speed counter/timers, digital communication protocols and pulse generation. Each channel is compatible with 5 V/TTL signals and features 1,000 Vrms transient isolation between the I/O channels and the backplane. The NI 9403 also features ±30 V overvoltage protection and can source up to 2 mA output current per channel [14].

Figure 2.5 NI-9403
2.6.3 NI-9263

NI-9263 shown in the figure below is a 4-channel, 100 k Sample/s simultaneously updating analog output module for any CompactRIO chassis. The NI-9263 also features ±10 V output range, 16-bit resolution, short-circuit protection, low crosstalk, high relative accuracy, fast slewrate and NIST-traceable calibration. The NI-9263 also includes channel-to-earth ground double isolation barrier for safety and noise immunity [15].

Figure 2.6 NI-9263
2.6.4 NI-9215

The NI-9215 shown in the figure below features 100 kSample/s per channel sample rate, ±10 V measurement range with 16-bit resolution, 250 Vrms channel-earth, CAT II (screw terminal), or 60 VDC channel-earth, CAT I (BNC) isolation.

The NI-9215 module can be used with both NI CompactDAQ and CompactRIO chassis. The module has four analogue input channels and a successive approximation register (SAR) 16-bit resolution analogue-to-digital converters (ADCs). The NI 9215 contains NIST-traceable calibration, a channel-to-earth ground double isolation barrier for immunity, and high common-mode voltage range [16].

![NI-9215 Module](image)

*Figure 2.7 NI-9215*
2.6.5 NI-9871

The NI-9871 consists of four RS232 or RS485/RS422 serial ports respectively. With the use of LabVIEW FPGA API, you can access the four ports directly from the CompactRIO FPGA to achieve flexibility in communicating with serial devices. To save space and simplify the programming for the CompactRIO FPGA, each individual buffer on every port is 64 B. The C-series modules support standard start bit, stop bit and handshaking settings and feature baud rates up to 921.6 kbit/s per port. It is possible to pass up to 2 Mbit/s of data between these modules and CompactRIO [17].

![Figure 2.8 NI-9871](image)

In this thesis, I have been given access to experiment with all the C-modules except the one that manages serial communication. I even borrowed a CompactRIO chassis and a computer with LabVIEW software from National Instruments to perform the tests. I would once again thank National Instruments for all the instruments and devices that I have got the opportunity to borrow.
3  Theory

3.1  Development of a Novel Low-Cost Flight Simulator

The following chapter will go through an environment study on how the development of a Novel Low-Cost Flight Simulator is carried out.

3.1.1  Introduction

The growing use of flight simulators for pilot training by major airlines is universal, and its effectiveness is well recognized. The problem is that the availability of the flight simulators are very restricted which means that it cannot meet the huge training needs of all airlines due to the high purchase cost, deployment cost and per hour usage cost of a traditional simulator. So the airlines are seeking affordable ways to train their pilots which means cheaper simulators that can cover certain training tasks.

Recently a novel simulator called ”light flight simulator” was developed in a laboratory by a college of civic avionics in China. This simulator is heavily simplified to reduce its cost and weight. This is done with the help of adopting several new technologies, such as mixed reality technology. The mixed reality technology is used to build up a semi-virtual cockpit which has a realistic tactile feeling by equipment of true instruments, and also a realistic stereo visual feeling by computer synthetic scenery and helmet monitors. This will lead to a reduction in overall weight, volume and cost of the visual system and make the cockpit design more simple. The novel simulator is based on a 6 DOF (6 Dimensions Of Freedom) motion platform can make the whole simulator accommodate in a normal office room [1].
The architecture of the novel flight simulator can be studied in figure 3.1. The structure of the simulator can be described by:

- The visual subsystem provides visual cue.
- The cockpit subsystem including all instruments provides tactile cue.
- The control loading subsystem provides force feedback.
- The motion subsystem provides vestibular cue.
- The acoustic subsystem provides auditory cue.

### 3.1.2 Aircraft Dynamics Mathematical Model

A necessary component of the simulator’s software is the mathematical model, which calculates the aircraft’s response to control inputs at various speeds and orientation. After that the output signal is transmitted to other subsystems to generate visual scenery, motion, sound and force feedback feelings for the user.

This simulator does not have a in-house development for the flight dynamics engine. Instead, they have decided to make use of pre-existing software called JSBSim for the following three reasons:

- It is an open-source software which can lower the overall cost.
- It is highly configurable and can be easily reconfigured by a XML configure file as a different aircraft type
- It can be modified to meet our specific demand and can be added some special effects such as turbulences, for the reason that the whole source code is totally open.
3.1.3 Motion System

The motion system is used to simulate the motion of the aircraft and provide the pilot with realistic feelings and a part of tactile feelings. According to the regulations given from FAA (Federal Aviation Administration) any device being called “flight simulator” must have at least one motion system, otherwise it should be called “flight training device”. From Fig.1, you can see that the motion system consists of motion platform, motion transformation algorithm, motion protection, washout filter and motion controller.

The pilot can feel the status of the aircraft by platform motion and force feedback given by the control loading system, thus provides corresponding control command input. After that the aircraft dynamics mathematical model can compute reference motion of aircraft according to varying control input. To drive the motion platform they have a "washout filter" to transform the aircraft reference motion signal to the platform reference motion signal. The washout filter works in the way that it divides the aircraft motion signal into two parts. One part is on high frequency domain can drive the platform directly and provide transient motion feeling for the user. The other part is on low frequencies and needs to be simulated by so called platform-tilt technique, which makes use of gravity component to simulate the continuous acceleration to generate the platform motion signal.

The platform is controlled by six high-power torque motors to drive the whole platform [1].

Figure 3.2 Motion platform without cabin
3.1.4 Semi-Virtual Cockpit System

The cockpit design of the novel flight simulator is different to other conventional flight simulators. The traditional cockpit of a flight simulator is almost a replica of the real aircraft and generally has a wide view projection system to generate high quality virtual out-of-window scenery. This means that the pilot do not have to wear any accessories during flight testing.

But this novel flight simulator is made to fit an office and meantime be cheap so they have come up with another way to solve it.

The wide-view projection system is huge and expensive and the simulated cockpit can only be used for one aircraft model.

To solve this they use Virtual Reality (VR) which simplifies the design of the cockpit system and visual scenery. The pilot use a helmet monitor to provide both cockpit scenery that includes appearance of all instruments and out-of-window scenery.

There is also two cameras on the helmet mounted near the pilot’s eyes which is used to capture the image of hand manipulation. Then the hand images is merged in to the virtual scenery which is displayed on the helmet’s monitor.

This makes the simulator flexible and easy reconfigurable to another aircraft type. Another great aspect is that the development cost is cheap for the reason that many functions can be realized with software instead of expensive hardware [1].

*Figure 3.3 The inner cockpit and out-of-window scenery*
3.1.5 Control Loading System

When the pilot induct a control input on a real aircraft, the aircraft’s control structures always provide a counterforce as the control mechanics carry out that input.

In a common flight simulator, hydraulic control systems is applied to produce proper forces on pilot control systems. The problem with the hydraulic control systems is that it maybe will encounter system failures such as hydraulic leaks, and the building and maintenance cost is relatively high.

For the novel flight simulator they have developed the control loading system in another way. Instead of hydraulic system they use an electronic driven control loading system to provide counter forces to pilot control inputs [1].

3.1.6 Other Subsystems

Other important subsystems are acoustic system, communication system and control console. The communication system connects all the subsystems with UDP protocol and in the same time manages the frame rate of the whole system. The base frame rate of the system is around 100 Hz.

The acoustic system consists of sound database module, sound generation module and sound display module. The sound database modules has the sources of engine noise, tire noise, wind noise and other background noises [1].

3.2 Development of a Pilot-in-the-loop Flight Simulator using NASA’s Transport Class Model

The following chapter will go through an environment study on how Nasa have developed their Transport Class Model.

3.2.1 Introduction

This operating environment study will focus on the development of an immersive flight simulator at the University of Illinois using NASA’s Transport Class Model (TCM).

The Flight simulators are used extensively throughout the aviation industry to evaluate system performance and a great tool for system design, verification and validation (V&V) of the system [2].

Verification and validation, used to verify that the system is designed correctly and working in the intended manner is a crucial step in the development of any safety-critical system, and many software tools exist for system V&V. A complement to these software tools is the process called human-in-the-loop testing (HIL), or pilot-in-the-loop (PITL). Human-in-the-loop testing is primarily useful for testing subjective aspects of the system, such as the “feel” to the human operator, that cannot be determined by the software methods.

Simulators are used for both hardware design and software testing, as well for pilot training and performance evaluation [2].
The PITL simulator at ISL consist of a Frasca 142 cockpit [6] and three projectors that contribute to a 180° view with graphics driven by X-Plane 9 [7]. To communicate between the computers in the system they use User Datagram Protocol (UDP) to drive the aircraft dynamics, cockpit controls and the aircraft dynamics.

### 3.2.2 TCM Simulation Model

The TCM simulation was further developed from the base of an earlier simulation technology called GTM, which is an abbreviation of Generic Transport Model. The GTM simulation was implemented with Simulink software which is a simulation tool from Matlab [4] environment. So to make things easier, the TCM simulation was also implemented with Simulink [5] software.

With this software, the simulation is constructed using hierarchical windows shown on the computer display. The simulation system is divided into different levels, the top level for instance represents the overall system with all of its sub-systems. To get to a lower system level you just press one of the sub-systems, the lower you get in the hierarchy the more detailed information you will get.

The Simulink model of the TCM aircraft was developed by NASA for research and development of flight control laws to prevent Loss of Control (LoC) incidents and manage upset conditions that could potentially lead to an LoC event [3].

The top level Simulink diagram for the TCM simulation is shown in figure 1. The system block called GTM_Full Scale contains subsystem blocks that contain aircraft dynamics, surface actuator dynamics, engine dynamics, aerodynamics model, aircraft system models and sensor models which is represented in figure 2.

![Figure 3.4 Top-level Simulink block of TCM simulation](image)

*Figure 3.4 Top-level Simulink block of TCM simulation [2]*
For example, the GTM_FS_Actuators block calculates the positions of the control surfaces using the mathematical actuator dynamics model. The information that is the output from this block is then available to other subsystems through the use of a global signal routing tag, shown in orange in Figure 2.

GTM_FS_Engines, GTM_FS_Aero, Gravity and Landing Gear are blocks used to calculate forces and moments with mathematical models generated by the engines, aerodynamics, gravity and landing gear. The forces and motions is then routed to the EOM block (equations of motion) along with the aircraft geometric and inertial information from the A/C parameters block. The next block called Auxiliary Variables calculates the aerodynamic variables such as angle-of-attack, angle-of-sideslip, true and estimated airspeed and dynamic pressure. GTM_FS_Sensors provides feedback to the cockpit instruments.

### 3.2.3 Simulator Hardware

The flight simulator that are used in Illinois Simulator Laboratory uses a modular architecture to administer a flexible simulation environment where components can operate independently and communicate over local area network using UDP protocol [2].

The schematic architecture shown in Figure 3 is made to be versatile and can therefore be applied to a variety of simulator hardware and software, and the tools developed can be used in other flight simulators, visualization software or aircraft dynamics models.
The Frasca 142 cockpit that are used in this simulator is equipped with a yoke-wheel and pedals as well as controls for the throttle, flap deployment, spoiler deployment, trim settings and additional features available if needed for the different aspects of the simulation. All button and control signals from the cockpit are then read through an ADC card connected to a computer and sent via User Datagram Protocol over the local area network to be used in the dynamics model driving the simulation [2].

The graphical representation of the entire system is driven by four computers and they are all running X-Plane 9. There are three displays that together contribute to a 180° view with graphics and one projector for each display. So there is three computers for the displays and the fourth main computer runs the master copy of X-Plane 9.

This main computer has the aircraft dynamics built into X-Plane, used to provide the simulation experience. From the main computer, aircraft position and orientation is sent via UDP protocol to the other three computers to update the graphics as new information is received.

Figure 3.6 Schematic representation of the simulation at the Illinois Simulator Laboratory

Figure 3.7 The flight simulator cockpit and graphics view at the Illinois Simulator Laboratory
3.3 **Overall System Description at Saab**

A brand new I/O-System shall be developed using COTS (Commercial-Off-The-Shelf) products and educational transference possibilities.

### 3.3.1 System Description

The system description will be based on COTS products. An overall view of the functional chain for the I/O-System is presented in the figure below. To consider in the picture below is that A/C Simulator is just another name for ACSIM that you may have noticed earlier.

![Figure 3.8 System Architecture](image)

The base component for the system will be National Instruments CompactRIO.

As previously stated in the report, CompactRIO is a reconfigurable embedded control and acquisition system. The rugged hardware architecture that CompactRIO has includes I/O modules, a reconfigurable FPGA chassis and an embedded controller. CompactRIO can be used in a variety of embedded control and monitoring applications, due to CompactRIO is programmed with NI LabVIEW graphical programming tools.

Down below is a more detailed description of the functional chain for the NI I/O-System.

![Figure 3.9 NI I/O System with MXI Express](image)

If it is required to use a larger number of I/O-signals, a NI CompactRIO with MXI Express support can be used and if that would not be enough nearly any PXI embedded controller or industrial PC with MXI Express can be used instead.

The picture below show a simple system description of the functional chain with MXI Express support.
3.3.2 **CompactRIO**

CompactRIO consists of the following components, an embedded controller for communication and processing, hot-swappable I/O-Modules, a reconfigurable chassis housing, the user-programmable FPGA and a graphical LabVIEW software for rapid real-time programming.

The following is represented in figure 3.10 as:

- Controller I/O SW
- User-programmable FPGA I/O FPGA
- Hot swappable I/O-Modules I/O HW

When it was time to choose a suitable controller for Saab’s purposes, following criteria’s were used:

- ”Many” slots for hot-swappable I/O-Modules
- Two Ethernet connectors (RJ45)

The components discussed below is not the same as I borrowed from National Instruments to perform my tests, but they work in the same way.

The best choice for the section at Saab that works with the development of simulators is therefore the NI cRIO-9067, with two Ethernet connectors and eight slots. Digital I/O will be managed by NI-9403 which has 32 digital input/outputs.
The analogue input signals will be handled by NI-9205 which has 16 differential inputs. The analogue output signals will be handled by NI-9264 that support 16 channels.

With the following configuration above, the cRIO-9067 will support as many as 256 digital I/O-signals and 128 analogue I/O-signals.

The reason why it has to be two Ethernet connectors derives from the design, one channel for the simulation and one for administration.

For much larger systems a NI cRIO-9081 or cRIO-9082 can be used in association with one to five expansions chassis with the maximum number of six chassis chained with cRIO, according to NI, PXI has the capabilities of more. A suitable chassis for this purpose would either be NI-9157 or NI-9159, which both have 14 slots and would through the MXI-Express bus support a higher number of digital I/O-signals and analogue I/O-signals.

A simple calculation on the aircraft Computer I/O-System means that it would be required to use one NI cRIO-9081 and at least three NI-9157 to accommodate the current signal number.

\[
\text{Digital I/O: } \frac{276!}{189!} \approx 20 \text{ slots}
\]

\[
\text{Analogue Inputs: } \frac{9!}{32!} \approx 5 \text{ slots}
\]

\[
\text{Analogue Output: } \frac{93!}{28!} \approx 25 \text{ slots}
\]
3.4 Design Decisions

The following section will go through the design decisions.

3.4.1 Main Timing

The figure above shows how the timing is supposed to look like.

The desirable design above, applicable to all I/O-Systems (old I/O-System, NI I/O), is that sampling of all the inputs, performing all the calculations required and also have time left to set the outputs before it is too late and you end up out of sync.

Because it is a real-time simulator, the entire chain of reading, performing calculations and setting outputs should be done within a 100 Hz frame.

3.4.2 Supported Signals

BIT

Bit signals will be handled default by NI-9403 that supports 32 digital inputs/outputs, $7 \mu s$ bidirectional digital I/O. You can configure the direction of each digital line on the NI-9403 for input or output. Each channel is compatible with 5 V/TTL signals.
**Float and integer**
Float and integer inputs will be handled default by NI-9264 that supports 16 channels, ±10V output range, 16-bit resolution, and 25 kSample/s per channel simultaneous analogue output.

**Serial Communication**
The serial communication will be handled by the NI-9871
The NI-9871 consists of four RS232 or RS485/RS422 serial ports respectively. With the use of LabVIEW FPGA API, you can access the four ports directly from the CompactRIO FPGA to achieve flexibility in communicating with serial devices. To save space and simplify the programming for the CompactRIO FPGA, each individual buffer on every port is 64 B. The C-series modules support standard start bit, stop bit and handshaking settings and feature baud rates up to 921.6 kbit/s per port. It is possible to pass up to 2 Mbit/s of data between these modules and CompactRIO.

**ARINC 429**
ARINC stands for Aeronautical Radio Inc. The ARINC 249 specification determines how avionics equipment and systems communicate on commercial aircrafts. The specification defines word structures, electrical characteristics and protocols necessary to establish bus communication. ARINC 429 utilizes the simplex, twisted shielded pair data bus standard called Mark 33 Digital Information Transfer system Bus [18].
4 Method

4.1 Implementation and Experimentation

In the following chapter I will describe how I have proceeded to verify and validate the NI I/O-System.

4.1.1 Brief Introduction of the various test cases

The figure above shows a schematic diagram of how the whole system looks like. As one can see in the figure, we would like to perform a test between A/C-Simulator and I/O-System to see how the signals go. This thesis will examine in detail how four different tests have been applied to the I/O-System using LabVIEW software to verify and validate that the system works as intended.

The first test that I will go through will verify that the hardware, borrowed from National Instruments works as intended before applying it to the whole system. This is done because I want to exclude a factor if the whole system later becomes erratic during testing.

The second program that has been done is a program that tests the UDP protocol in LabVIEW to validate that the packages you send arrive as they should. Therefore, It means, a separate program consisting of smaller sub-programs. The first program creates a package, then the second sub-program read the packet, to finally get printed by the third program.

The third test will test the entire I/O-System from beginning to end. This is done to verify that the I/O-System works as intended. Because the I/O-System sends all the information via UDP protocol, I created a similar program as the one talked about previously that will read and write data. It will consist of two programs cooperating with each other using IP
communication. The first program is the I/O-System and the second is the UDP program. The UDP program will in this case simulate how the A/C-Simulator works in a simple way, because in the current situation, it will not be possible to test the entire chain as mentioned in the previous figure. How this works will be further described in detail later.

For this test, It was actually the simulator section at Saab themselves who came up with the idea. They wanted me to get a deeper understanding of the I/O-Nodes and solve one of their problems they have encountered after that. The problem here was that the people who where working on the development of simulators where wondering what happened to the signals after you have read them from the I/O-Modules and stopped everything in an array in LabVIEW.

The idea behind this test is to verify that the signal order of an array is consistent through the entire chain. One should be able to set different bits of the output signal with this program and check if that one bit has actually changed and were it is in relation to where it was.

Lastly, I will go through how to make a test bench VI to increase productivity and reduce development time for LabVIEW programs. LabVIEW FPGA provides the user with simulated I/O which executes your application on your development computer instead of compiling the program and running it on the FPGA which takes long time for large and complex programs.

4.1.2 Setting up a new project in LabVIEW

The following tutorial below will show how to start a new LabVIEW project and will give the basics on how to get started with the program. Depending on the hardware you have, the tutorial will be different, this tutorial will show how to set up a project with the same hardware that I used to perform the tests in this thesis.

This first part will be the same for all future project that I will talk about later in this report.
1. Start by launching **LabVIEW**

2. In the **Create Project** screen in LabVIEW, select **Blank Project**. The Project Explorer window will now appear.
3. Select **File>Save Project** and enter the name for your new project after selecting the appropriate directory. I choose tutorial.lvproj.

4. In the **Project Explorer** window, **Right-click on Project: (ProjectName).lvproj** and select **New>Targets and Devices...** The **Add Targets and Devices** window will appear.
5. While **Existing target or device** is selected, select the appropriate type of device you have connected and LabVIEW will gather all hardware on its own. Otherwise press **New target or device** which will give you the opportunity to type in hardware manually. This is a great way to simulate the hardware which enables the user to test the program without having to purchase expensive hardware first. The following tutorial will show how to set up a project manually.

6. Select **Real-Time CompactRIO and select cRIO-9022**
7. Right-click the RT Target and select **properties**. Select **General** and type in the following IP-adress: **192.168.0.100**

8. Right-click the RT-Target and select **New Targets and Devices...** Select **cRIO-9114** chassis.
9. Right-click **Chassis** and select **New FPGA Target**

10. Right-click **FPGA Target** and select **New C Series Modules**… Then select **New target or device**. Add **NI-9263** for slot 1, **NI-9215** for slot 2 and **NI-9403** for slot 3.

11. Right-click on **FPGA Target**, select **NEW VI** and type in your name for the new VI. I prefer to call it FPGA_Main.vi

12. Right-click on **My Computer** and select **New VI** and select a name for your VI. I prefer to call it Host.vi
4.1.3 Complete NI I/O-System Overview

The following figures below show how the entire I/O-System is built in LabVIEW. According to the tutorial previously discussed in the report, the code will consist of a Host and an FPGA code. In the tutorial, the Host code is located on the desktop environment but in this case it shall be under Chassis (cRIO-9114) environment. The first picture shows how the code is located on the FPGA level is designed. The second picture shows the code that act Host for the whole system looks like, at the top of the hierarchy, in other words.

![Figure 4.2 FPGA Code](image)

The code located on the FPGA level shown in the figure above consists basically of two I/O-nodes that have the task to manage the input/output of the various C-modules that are connected to the cRIO-9022 chassis.

The FPGA code is divided into two case structures, one called the input task and the second one called output task.

The input task has the assignment of taking all the incoming signals from the NI 9215 and NI-9403 modules and remodel all the signals to integers and then stop everything in an array to be sent upstream.
Output task has the responsibility to read the data downstream, split the array into single signals again and then set the output node. The figure above is also located in the appendices for further understanding.

Figure 4.3 Host code

The code located on the Host level, has the task of receiving and sending packets from the I/O-System. First of all, you have to put a FPGA reference in the beginning as you can see in the code above that consists of the code on the FPGA level to tell which target you want to run the Host Code against. Mainly the code basically just consists out of different FIFOs which means First In First Out, FIFO acts thus like buffer memory. The code then consists of two smaller case structures, one called packet transmission, and the other one is called Packet Reception.

Packet transmissions task is to read the upstream data coming from the FPGA code and then via UDP, send the packet forward up to the simulation computers that runs the simulation environment.

Packet Reception has the responsibility of receiving packets coming downstream from the simulation computers via UDP communication.

The images shown above may be difficult to study in detail because of the size of the graphical programs. Therefore, I have included the pictures as attachments in the appendices further back in the report in a larger size for further studying.

4.2 CompactRIO Hardware Test

The first test I did was a test that verified that all hardware that was connected to the computer functioned as it should. I did this by testing the internal resistance of the National Instruments hardware.

Later i will test the entire I/O-System and if wrong data would appear, I can be sure that nothing is wrong with the hardware after this test.

With this test, I will also find out how much the signals will change between the I/O-Modules.
The easiest way to get started with this test is to follow the structure that I have already gone through in the *Setting up a new project in LabVIEW tutorial*.

The easiest way is to start with the code that should be located at the FPGA level and later on finish the Host Code.

The hardware available is CompactRIO and three I/O-Modules that will manage analogue and digital input/output.

Analog input will be managed by the NI 9215 and analogue output will be managed by the NI-9263. The modules have four channels each.

The digital signals will be processed by NI-9403 that works in a different way, both input and output is done on the same module, which means you can configure the ports yourself to read or write in the software.
4.2.1 FPGA Code

The figure below shows how the code at the FPGA level looks like for this test. The code basically consists of four different I/O-Nodes as it is called in LabVIEW and each node is a graphical representation of the various I/O-Modules which are connected to the CompactRIO chassis. "Mod1" and "Mod2" represents the analogue signals and "Mod3" represents the digital signals.

For instance, "Mod1/AO2" is represented by I/O-Module in slot one on CompactRIO chassis with signal port two.

The user has to choose whether the digital I/O-Nodes should read or write by clicking on the signal to be changed.

Figure 4.4 FPGA Code
4.2.2 Host Code

The Host code is in the top of the hierarchy and is the last step of the development in this test, shown in the figure below. As earlier explained, the host code begins with setting the target to CompactRIO so LabVIEW know which hardware you are working with. After that, you have to put a FPGA reference in the target as you can see in the code below that consists of the code on the FPGA level to tell which target you want to run the Host Code against. The code consists of a while-loop that will execute until the user presses the stop button located at the front panel that will be generated of the global variable "True/False", as you can see in the figure below "status".

In the while loop, there is also a "Read/Write Control" with a control on each output which allows the user to set their own output values from the desktop environment.

By doing this, the user will then be able to go down on the FPGA level and check if the actual value that was set as an output actually becomes the same value at the input.
4.2.3 Performing the test

Below is a consistent explanation of how to test the program to verify that the hardware actually works as intended.

The images below show how the front panel looks like for the Host and FPGA code. The front panel is generated by the code depending on which controls and indicators used. From the front panel, the user can control the code that is currently under execution.

Figure 4.7 Host Front Panel

Figure 4.8 FPGA Front Panel
To verify that the hardware works as intended, you put a reference value to the output you want to test on the front panel of the Host. Then you go into the front panel of the FPGA code to see that the signal actually arrived at the input and how much difference you will get.

For instance, if an output at the front panel is set to generate 2 V, we will receive approximately 1.997 V at the input, which means you will not be able to get the exact value you want.

This will result in a signal loss between the I/O modules at 0.15%.

\[ \frac{1.997}{2} = 0.9985 \]

### 4.3 UDP Communication Test

The following section will go through a UDP Communication test.

#### 4.3.1 Overview

The following program will show the basis for the test that will be performed on the entire NI I/O-System later. This program is designed to test the UDP communication in LabVIEW to get deeper understanding of UDP before testing the entire system. This test checks that the UDP communication works as intended and is very important, since the communication between NI I/O-System and the simulation computers is done with UDP.

#### 4.3.2 Implementation

The figure below show an overview of how the test is designed in LabVIEW. The program is structured in three sub VIs located inside a flat sequence structure, which in turn is inside a case structure. I am using the Flat Sequence structure to ensure that Send package UDP executes before the UDP Read and that the UDP Read executes before the UDP Write.
First of all we open up UDP communication with the Sub VI shown in the figure below. UDP Open attempts to open a UDP communication on the port specified by the user. Connection ID is an opaque token used in all subsequent operations relating to the connection.

![Figure 4.10 UDP Open](image)

The sub VI called UDP Close tries to close the UDP connection specified by the connection ID out. Connection ID is a network connection refnum that uniquely identifies the UDP socket you want to close.

![Figure 4.11 UDP Close](image)

The first sub VI called Send package UDP.vi will create a package and then through UDP protocol send the packet upstream and write the data on the specific port number that is specified by the user. Then it is important to enter the same port number as you entered in UDP Open, in this specific case port 1026. The package created below is a package containing upstream data that in this particular case will be empty and not contain anything, as it will be used for a subsequent test case for the entire I/O-System later on. So here you have to manually decide what value it will receive.

The package will then consist of a unsigned 16-bit Card ID, unsigned 32-bit Iomap-ID, unsigned 8-bit system ID and a counter consisting of Frame ID MSB and Frame ID LSB. Frame ID MSB is unsigned 16-bit and frame ID LSB is unsigned 32-bit. The code is hard-coded, which means creating a package in this step is done all manually. So you have to click yourself into the Header and set fixed constants, because this program is designed to work with the real I/O-System later, which means that the values will be set automatic instead.

![Figure 4.12 Send Package UDP](image)
The code that can be studied in the figure below is the second sub VI program called UDP Read.vi. This program will read the packet sent from the earlier program called Send package UDP.vi, and then create an array of the information called Downstream data as shown in the picture below. Before the program creates an array of the information that was received, it has to go through some tests to verify that the package is correct.

The controls consist of three case structures that must be fulfilled for the information to be sent to Downstream data.

The controls are Iomap-ID, system ID and Downstream words used to verify that the package is correct. The idea of the system ID is to tell the program what the version ID the current I/O-System has, system ID will therefore change depending on the configuration running the system. If you update the I/O-System with new hardware, then system ID will also change.

Iomap-ID is the identity of the package, each packet gets a new identification number.

Downstream words specifies the size of the array coming from the I/O-Nodes and it must be less or equal to the size of the array.

![Figure 4.13 UDP Read](image)

The third program is more or less the same as the first program called Send Package. The only thing I do in the last program is to look at the array I created in Send Package UDP.vi, so it arrives in the right order and is printed properly. I will show the front panel for this code later in this section.
The figure above show how the front panel for Send Package.vi looks like, here the user creates the package that should be sent.

Upstream Data is filled in manually with values to create an array of information. This is done so I later on in the program can go in and see if the array looks the same when it has been printed by UDP Write.vi.

Then I add values to the Header In with information I can send off to the other VIs and through the chain, control that the values are changing. The reason that all values is being entered manually at the moment, is because the code is designed to work on the I/O-System later on which means that these values automatically will change. For example, Frame ID MSB consists of a counter that is not currently used, it will be used later on the FPGA counting up every time the program sends a signal from the I/O-Modules.

The figure below show you the program UDP Read.vi in the middle of an execution run. By using the probe function built in by LabVIEW when running the program, you can study all the signals values under the execution stage, and if you want more information about each individual signal, just press them using the probe. A very good feature in LabVIEW that has been used frequently to troubleshoot when problems arose.

Here one can see the different control stages in action during execution. The figure below show that this package is correct and therefore creates an array of seven dimensions sent to Downstream data.
The figure above show how the front panel of the UDP Write.vi looks like, the front panel is used as a control to ensure that the package that was created in Send Package.vi have arrived in the same order as it was sent. In the picture above you can see that the array that was created in the Send Package.vi has been read by the UDP Read, then sent downstream data to the UDP write for print in the correct order.

### 4.4 Implementing UDP Test on NI I/O-System

The following section will go through a UDP Test on NI I/O-System.

#### 4.4.1 Overview

This test is made to implement the UDP Test code on the NI I/O-System and run them both at the same time to verify that the system works as it should. Since the development of the new simulator has not come far enough yet to run the entire chain from simulation computers
down to the I/O-System, the UDP Test will simulate a simple version of a real simulation computer, in order to perform tests on the system.

4.4.2 Implementation

This test will consist of two different LabVIEW program that will run in parallel to each other while they are communicating via UDP protocol. One program will consist of the I/O-System that will run the code, create an automatically generated packet and wait to send the packet upstream to port number 1026 through UDP communication.

The second code that I have chosen to call for UDP Test code will run simultaneously as the I/O-System code. This code sits and waits to open up UDP communication on port 1026, which is the same port that the I/O-System will send its packet upstream to. When the communication on port 1026 has been opened, the package that has been sent upstream from the I/O-System will be read by the UDP Test code. After the package has been read, it will be sent back downstream to the I/O-System and finally printed.

Simply stated, the I/O-System will create a package, send it to the simulation computer which in this case is the UDP Test program and then sent back downstream to the I/O-System again to be printed.

The test is successful if all information created in the I/O-System will arrive in the same order as it was sent.

The test will thus verify that the packets sent via UDP actually arrive correctly.

![System Architecture Diagram](image)

*Figure 4.17 System Architecture*

The test is thus designed to simulate the communication between the A/C Simulator and I/O-System that can be seen in the above figure.
The figure above shows a picture of how the code for the UDP Test program looks like for this test case.

This code will thus stimulate the NI I/O-System to see if it works as intended.

The code is based on the previous test case when only UDP communication was controlled without the I/O-System. The difference here is that this program does not create any packages by itself as the former did.

This code will constantly sit and read packages all the time and then send that information downstream via UDP protocol.

In this case, the package will be created in the I/O-System program, which means there is no longer needed to use the third block called Send Packet.vi used in the previous program.

In the program above, "Receive Port from Packet Transmission 2" is used to say which port the communication between the different programs should be made, in this case, port number 1026. "Downstream Port to Packet Reception 1" will establish the downstream traffic on port number 1027.

This system will, as previously mentioned, simulate the communication between the A/C simulator and I/O-System. This means that the UDP Test code will act simulation computer and I/O-System simple hardware. The communication between the simulation computer and I/O-System requires that you specify which IP-address the packet should be sent to. "Downstream address" as you can see in the figure above, will therefore specify the IP-address of the CompactRIO hardware. This IP-address is 192.168.0.100 and will be used for all the packets sent downstream.
The figure above shows a small snippet of the I/O-System code that takes care of the UDP communication. The I/O-System thus consists of a packet transmission that creates a packet and packet reception that receives and prints out the package.

In the figure above, one can see that the code uses two UDP Open Vis. This is done because packet reception and packet transmission will manage communication between various ports, because one cannot read and write at the same port.

"Receive Port from Packet Testing 1" opens UDP communication for packet reception at port number 1027.

"Send Port to Packet Transmission 2" opens up UDP communication at port number 1026.

"Upstream Port to Packet Testing 2" will send the packet upstream to port number 1026.

"Upstream Address" will define which IP-address the packet should be sent to, in this case the IP-address is 192.168.0.2 for the simulation computer.

When you have set the IP-addresses and ports as the instructions above, the I/O-System will create a package and send it on to the simulation computer. Then the simulation computer will send the packet downstream to the I/O-System which in turn will print the information in the same way it was created.

This test does not show the whole truth, on the actual aircraft simulator, the simulation computers will execute models of the real aircraft hardware and I/O in real-time, which is not implemented in this test.

A figure of the entire I/O-System will also be featured in the appendices for deeper understanding.
4.5 I/O-Node Test

The section below will go through the implementation of a I/O-Node test.

4.5.1 Overview

It is for the new flight simulator, one has pioneered the use of CompactRIO, as it is a system that many around the world can use and have great knowledge around.

For the moment, however, there are many questions about LabVIEW and the I/O-System. A question that had to be answered where how to verify the signal order coming from an I/O-Node which in turn is added together to form an array in LabVIEW, and verify that the order is the same throughout the entire chain. This leads us to what this test is all about.

The test will also check what happens if you modify the signal by hand and set different bits into the array, if they will arrive in the same order as they were inserted, the test is green.

4.5.2 Implementation

The figure above demonstrates how the question mark at the simulator department at saab looked like. One therefore wondered what the blue signal contained before the Write FIFO.vi, after it has been merged together by all digital and analogue signals from the I/O-Node.

It is very important to keep track of how the signals go in the program because later on in the development, the program will be applied to the real aircraft hardware. Without the signal knowledge before applying it to hardware, major problems can arise and may cause hardware damage.

A requirement from the simulator department was that the program could be run entirely separately from the desktop environment, which means that you want to try this problems without having access to real hardware in place. If you have no real hardware available, you then have to generate all the signals to your I/O-Node on your own.
In order to generate the signal on your own, you must put the Execution Mode to Simulation instead of the FPGA Target. The figure above describes how to change the execution mode for simulation (simulated I/O) instead. In doing so, the signals from the modules are being simulated and will therefore give out simulated signals with random values.

To do this test, represented in the figure above, start by following the setting up a new project in LabVIEW tutorial described earlier in the report. The code is basically made of a flat sequence structure inside a while loop with I/O-Nodes in each ends. The left code handles all input signals and the right handles all output signals. A flat sequence structure is used here because I want the entire input chain to be executed and done before the output signals are set.
The figure below show how the code looks like in the first part that is executed first in the Flat Sequence Structure, thus the input signals. This code snippet below will pick up analogue and digital inputs from the I/O-Node. This time, however, no values from the CompactRIO I/O-Modules is generated because the hardware in this task will not be used. Instead I will use simulated analogue and digital inputs to the system to be able to run the whole program entirely from the desktop environment without any hardware connected.

This therefore means that the node will generate random digital and analogue values which are then put together into an array, which in this case is called "Output Array MOD2". Also available indicators of all the signals in order to be able to follow them precise signals from beginning to end during an execution in progress.

I have also connected controls between each signal from the I/O-Node to the array to give the user the opportunity to set their own values to the array. This is done in order to be able to set own single bits into the array and then at the output check that the bit is set and in the correct order.

Figure 4.23 Input Signals
The figure above represents the second part of the Flat Sequence structure, which will be performed after the input signal part has been successfully executed. The array that was created previously in the input part of the program was connected to an indicator. The values stated in the indicator is then stored away to a local variable that in turn gives the user access to the data for the output section too.

The local variable with the associated input values can be seen in the figure above as the "Output Array MOD2". To the local variable, I have also connected an indicator, used to verify that the signal order is correct.

The idea with this program now is that you can see if all the signals is consistent through the entire chain from input to output. And with the help of all the indicators, following each individual signal from the input node to the output node is now possible.

The controls that were set in the previous input part is now replaced by indicators to verify that the input bit that was set by the user earlier, matches the values coming out as outputs.
Figure 4.25 Host Code

The figure above shows how the Host code looks like with the FPGA code loaded and ready. The code consists of a Read/Write Control, which enables the user to look at what happens in the FPGA code with the indicators and to set own values to the FPGA code during execution from the desktop environment. The code also consists of a global variable as you can see in the figure above called "CANCEL EXECUTION" that will interrupt the program.
The figure above show how the front panel of the Host code looks like. The figure is taken after a execution has been carried out and one can see that all of the signals match through the entire chain, from beginning to end.

4.5.3 Accessing Real World I/O during FPGA Simulation Using CompactRIO

The following text will work as a guide to access real world I/O during FPGA simulation.

4.5.3.1 Overview

The following section will give you an example of how to increase productivity and reduce development time for LabVIEW programs. LabVIEW FPGA provides the user with simulated I/O which executes your application on your development computer instead of compiling the program and running it on the FPGA which takes long time for large and complex programs. By running the program on the development computer means that you can use all LabVIEW debugging features and you do not have to wait for the program to compile every time you change the code. By running simulation you will get simulated I/O but there is a way around
You can create a test bench VI to assert the inputs that would normally be connected to the outside world with user defined I/O and look at the outputs for verification and validation. There are two ways of accessing I/O in LabVIEW and they are FPGA Interface and Scan Interface. Scan interface automatically publishes the I/O values to the network which in turn makes it possible to create a test bench to read and write those values on the development computer and therefore getting access to Real-World I/O.

4.5.3.2 How to create a Test Bench VI to access Real-World I/O

![Diagram of System Overview Using CompactRIO Scan Mode for Real-World I/O](image)

The figure above shows the system overview of accessing real-world I/O using a test bench VI that reads and writes during FPGA simulation. When using Scan interface as execution mode, you get direct access to I/O in LabVIEW Real-Time without having to program the FPGA or compile it.

![Diagram of Overview of the Test Bench VI](image)

The figure above gives a brief overview of the Test Bench VI that establishes Real-World I/O connection to the development computer during an FPGA simulation. The figure above is compressed to fit the document but can be studied further in a larger format in the appendices.
4.5.3.3 Software Explanation

When setting the execution mode to Scan Interface, each I/O-channel in a module is published at psp://<IP address>/<Module Name>/<Channel Name>, where <IP address> is the IP address of the cRIO controller, <Module Name> is the name of the I/O-Module, <Channel Name> is the name of the I/O channel. So to explain this further, my project with my CompactRIO controller with IP address 192.168.0.100 with an analog output NI-9263 module in slot 1 will publish the following value: psp://192.168.0.100/Mod1/AO0.

4.5.3.4 Overview of the subVIs

Below is a description of the different Sub VI’s.

1. Returns the execution stage of the VI (Initializing, Running, Shutting Down)
2. Will return the name of the I/O item in the project
3. Will return the name of the node type (Read I/O, Write I/O)
4. Will return the name of the method or property
5. Reports that you have not specified behavior for a certain I/O item, node type
6. Passes any error to the LabVIEW FPGA Module

4.5.3.5 Overview of VI Inputs and Outputs

Data from node input terminals is an array of variants corresponding to all of the inputs of the FPGA I/O, Method or Property Node but only if the node has any inputs.

Data to node input terminals is an array of variants corresponding to all of the outputs of the FPGA I/O, Method or Property Node but only if the node has any inputs.

Input from node terminal error in is the error that gets passed into the FPGA I/O, Method or Property Node.

Output from node terminal error in is the error that gets passed out of the FPGA I/O, Method or Property Node.

4.5.3.6 Configuring CompactRIO controller to publish scanned I/O

Configuring the CompactRIO to publish the I/O values is simple, just do the following:

1. Drag the I/O-Modules in the LabVIEW Project Explorer from the FPGA Target to the Chassis.
2. Press the Chassis properties and select **Scan Interface**, and click OK.
3. Right-Click on the Chassis and select **Deploy**.

After the above instructions has been completed, LabVIEW will deploy the I/O-Modules and CompactRIO will begin publishing the I/O values. If you want to access real world I/O, drag
the I/O-Modules back to the FPGA Target and run your FPGA VI in simulation in parallel with your test bench VI.
5 Results

5.1 CompactRIO Hardware Test

Here I tested the hardware of CompactRIO to check if the signal value between input and output where consistent. The result here was a signal loss of 0.15% between the two I/O-Modules. For instance, if the output module on port 1 is set to generate precisely 2 V to the input module on port one, we will receive approximately 1.997 V at the input, which means that the value will differ from what the user wants. This will probably not affect the simulator department at Saab when they use the I/O-System. The problem here is that the equipment I have borrowed from National Instruments is not calibrated properly which will affect the outcome.

5.2 UDP Communication Test

The test here was to check the communication in LabVIEW using the UDP protocol. The test was successful and no error was detected in the packets that were sent.

First, the program opened up UDP communication, then created a package by a sub VI then to be read by a second sub VI, who in turn was printed by a third sub VI. The package that was created in the beginning was correct throughout the entire chain.

This test only shows that it is possible to send data via UDP protocol.

5.3 Implementing UDP Test on NI I/O-System

This test is made to implement the UDP Test code on the NI I/O-System and run them both at the same time to verify that the system works as it should. The I/O-System will create a package and send it on to the simulation computer. Then the simulation computer will send the packet downstream to the I/O-System which in turn will print the information in the same way it was created. The test was therefore successful.

5.4 I/O-Node Test

This test was about how to verify that the signal order coming from an I/O-Node which in turn is added together to form an array in LabVIEW, and verify that the signal order is the same throughout the entire chain. The test was successful and the signals were the same from input to output. One could also follow each individual signal from the input node to the output node. The bits that was set by the user matches the values at the output.
6 Discussion

6.1 Results

In general, the results have been as I expected but there is a few things I would like to discuss more about.

The first test that I conducted was to check that the CompactRio hardware that was available to my work functioned as it was supposed to. When I performed the test, I got a signal loss of 0.15% between the input and output module.

Later on in the development, when the time has come to implement the hardware for real, they will not screw the individual unshielded cables by hand to the I/O-Modules that I had to during the thesis.

The use of shielded twisted pair cables and connectors, used to reduce electronic interference between the cables may solve the problem.

6.2 Method

This method does not show a realistic picture of the system testing. To test the entire I/O-System for real, you cannot just sit with a computer program and simulate peripherals. Instead, to get a real-world test, you should have connected the external devices to the system while running the tests.

Without having any external devices connected to the system, you can never really know if it works as it should.

Something which I also did in the thesis but did not include in the report were measurements of latency in the system. This works without plugging in external devices and you get a picture of the system's performance.

I did this easily with a LabVIEW program where I divided the code into three sections. The middle section had the I/O-system itself and the others had timers. The program started by setting the timer, then executing the I/O-system one cycle and then stopped the timer when the program had finished executing.

Maybe I should have focused a bit more on this test and tried to get a picture of how much I/O that was possible to use without getting a bad result on the response time.
7 Conclusions

After completing the thesis and begin to look back to the purpose with the thesis, I feel that the whole process has followed a common theme, namely, to find a solution to a problem that has existed around the I/O-System here at the development department of the simulators at Saab. To find good ways on how to verify and validate the system for future use. The problem has been that the department does not know if the communication between the simulation computers and I/O-System worked as it should. It is this theme that the whole thesis work has revolved around, as my programs both have tested the UDP communication between the I/O-System and external devices, and also single tests to ensure that the signals inside the I/O-System arrives in the expected way.

I have answered the questions for this thesis through with the help of hardware borrowed from National Instruments and various testing using LabVIEW software.

The first question for this thesis was: **How should we proceed in order to ensure that the data transfer inside the input/output system is working correctly?**

My answer to this is that the I/O-System works as intended through the verifications and validations I have conducted in the way it is possible. This can be verified by running the LabVIEW program that implements UDP Test code on the I/O-System.

The system works with those simulations that has been conducted with the LabVIEW software. But it does not show a realistic picture of the results because it has not been tested with the real components connected to the system. This has not been possible to test in this phase of the development. The tests that I have performed, has only been the basis for developing ways to verify and validate the system before the company start using it for real. For a complete test, one would need to connect external devices, and run the entire chain from beginning to end. Which means connecting the I/O-System to a system Rig and connecting a SUT (Subject Under Test) to the end of the chain. This is the final test you must perform to get the answer of whether the system works as intended.

The second question for the thesis was: **How should we do to test so the format that talks with the I/O system is consistent with the real code?**

This question is answered in such way that the communication between the Simulation computers and the I/O-System works as intended at the level that it is currently possible to verify in the development process.

My test cases performed on the I/O-System with the help of LabVIEW software facilitates development and will answer some questions that previously were not answered.

Currently, the new CompactRIO I/O-system is a whole new area for the simulator department at Saab and it is not many people that know how it works in practice. To go from a product, originally manufactured by Saab themselves and then replacing it with a COTS product is not
an easy decision. Especially not considering that they have no previous experience of the new product from other aircrafts.

But I think that it is a great decision trying to move to COTS products. According to my way of thinking, COTS is the best choice in the end because it will lead to a higher aircraft developing pace. When they have COTS products to their disposal, they can get help from other specialized companies. Why it is so good to start using products that are fully developed and are already on the market is that many people already have a deeper understanding of the product, which in turn means that you can bring in outside help when it gets difficult.

This report could be used as a help for those who later on in the development will start working with LabVIEW and CompactRIO. As well as provide an overview of both the I/O-System and the entire flight simulator. Theory behind the hardware will also be available by looking at the report. The report will go through both the basics and advanced level of development for the flight simulator at Saab. This means that a person who is not the least bit familiar with the system before, should be able to read this report and afterwards have a greater knowledge and could hopefully start working for real.

If there had been more time for this thesis and the development had come further in the future, I would like to test the entire chain for real and see what the result is. Will the real test for the entire system later on give the same results that I had? If not, something with the system is wrong. If so, what would be the approach to solve the problem?
Bibliography


Appendix 1. Overview of the Host Code
Appendix 2. Overview of the FPGA Code
Appendix 3. Overview of the Test Bench VI