Julius Maximilian University of Würzburg
Faculty of Mathematics and Computer Science

Aerospace Information Technology
Chair of Computer Science VIII Prof. Dr. Sergio Montenegro





Master Thesis

Design and Implementation of a Failsafe Solution for Quadrocopters

Submitted by

Tobias Roos

Matr.-Nr.: 1845842

Examiners: Prof. Dr. Sergio Montenegro, Dr. Thomas Kuhn

Supervisor: Dipl.-Ing. Nils Gageik

Würzburg, 08. 10. 2013

Declaration

I hereby confirm that this Master Thesis, including all attached materials, was independently produced without using any sources other than those listed.
Any references to published or unpublished works, literal or otherwise, are clearly indicated and listed for each individual case. This work has not been submitted in identical or similar form as an examination paper.

Würzburg, 08. 10. 2013

Tobias Roos

Task description

The progress in microelectronics, sensors and actors enables nowadays the development of autonomous quadrocopters for plenty of applications. While outdoor scenarios are well researched, fully autonomous indoor systems are still challenging because of the lack of accurate sensors for positioning. Especially in the case of low cost and robust multi- environmental systems for exploration, which can overcome adverse environmental conditions like smoke, sensor failures are still a challenging problem for accurate positioning and control of a fully autonomous system.

The operation of an autonomous quadrocopter in such difficult environments is one goal of the AQopterI8 project, which is executed at the Chair of Aerospace Information Technology of the University of Wuerzburg. To increase the robustness of the quadrocopter in case of software and hardware failures, especially to overcome the negative effect of sensor failures, a failsafe solution has to be developed, which can detect software crashes and hardware blackouts.

In case of orientation and obstacle detection sensor failures, the quadrocopter is not capable to avoid a crash. Without continuous control crashes of the flying system within seconds canât be avoided. Therefore the task of this thesis is the development of a Failsafe System, which is capable to overcome hardware and software failures in real-time. The focus of this thesis is the usage of the operation system RODOS to realize a failsafe system for quadrocopters. With an intelligent electrical circuit, it shall be possible to keep a minimal amount of sensors and connections, while faulty components can be disconnected. Even a fatale error of the main processing hardware during the flight shall be possible. The construction and programming of the basic functions of the quadrocopter are not part of the task and are already given (quadrocopter software & hardware). Part of the task is an evaluation of the failsafe system under real-flight conditions and documentation. Task (in short):

- Background and Concept: Failsafe System for HW/SW Failure
- Realizing Failsafe Software using RODOS
- Intelligent Electrical Failsafe Circuit
- Evaluation in real-flight conditions
- Documentation

Abstract

The usage of small unmanned aerial vehicles has seen a steady increase in recent years, branching out from pure military applications to everyday use. Currently, research is performed at the Chair of Computer Science VIII at the University of Würzburg into an autonomous quadrocopter capable of navigating indoors through potentially hazardous environments. Until now, the system has had only limited redundancy in the form of multiple sensors and sensor fault detection. The goal of this project is to provide a failsafe solution that will enable the quadrocopter to continue operation in case of a failure of its central microcontroller.

The solution presented in this thesis uses a modified version of the quadrocopter's original software in combination with the RODOS real-time operating system to provide a software capable of redundant operation across several nodes. This is combined with duplex hardware redundancy and built-in testing to provide the full redundancy solution.

Results show an excellent failure detection rate; however, the secondary node often fails due to problems with the sensor and motor communication bus following a primary node failure. The implemented solution provides protection aginst failures of the central microcontroller and demonstrates the potential of using the RODOS operating system in the context of quadrocopter redundancy, but further work on the communication bus is required in order to provide a fully stable redundancy solution.

Contents

1 Introduction **1**

2 State of the art **3**

2.1 Current state of the quadrocopter project 3

2.2 The RODOS operating system 6

2.2.1 Operating systems and multi-threading 8

2.2.2 Data access 9

2.2.3 Data structure 11

2.3 Hardware redundancy solutions 12

2.3.1 Built-In Testing 13

2.3.2 Dual redundancy 14

2.3.3 Triple redundancy 16

2.3.4 Redundancy requiring more than three redundant units 18

2.4 Software 20

2.4.1 Temporal redundancy 20

2.4.2 Software diversity 20

2.5 Examples of fault tolerant systems 24

2.5.1 Cascaded TMR 24

2.5.2 Stateful TMR 25

2.5.3 DMR checkpointing 26

2.5.4 Asynchronous checkpointing and rollback 28

3 Concept **29**

3.1 Restructuring, multi-threading and RODOS 29

3.1.1 Task structure 31

3.1.2 Data structure 34

3.2 Redundancy 37

4 Implementation **41**

4.1 Software 41

4.1.1 First iteration 41

4.1.2 Second iteration 44

4.1.3 Third iteration 49

4.1.4 Final iteration 54

4.2 Hardware 57

4.2.1 Communication 57

4.2.2 Redundant sensors 59

4.2.3 BIT via relays 59

5 Evaluation **61**

5.1 Final iteration, non-redundant hardware 64

5.2 Final iteration, redundant hardware 65

5.3 Additional testing 69

6 Discussion and outlook **70**

7 Bibliography **73**

List of Figures

1.1 The AQopterI8. 2

2.1 Overview of the initial quadrocopter functionality. 4
2.2 Overview of the main loop of the quadrocopter software. 5
2.3 Overview of the data structure and inputs/outputs of the quadrocopter software modules. 7
2.4 Comparison between single-threaded and multi-threaded designs. 8
2.5 Example of thread scheduling. 9
2.6 Single-threaded data access. 9
2.7 Multi-threaded data access. 10
2.8 Inter-thread communications. 12
2.9 System without fault-tolerance. 13
2.10 System with built-in testing. 13
2.11 System with dual redundancy. 14
2.12 System with dual redundancy and BIT failover. 15
2.13 System with dual redundancy as a self-checking pair. 15
2.14 Self-checking pair with fault-down capability. 16
2.15 Triple modular redundancy 17
2.16 4-channel N-modular redundancy. 19
2.17 Dual self-checking pair. 19
2.18 Checkpointing and rollback. 21
2.19 Software diversity in a single hardware unit. 21
2.20 Software diversity in multiple hardware units. 22
2.21 Software TMR. 22

2.22 Software and hardware TMR. 23
2.23 Cascaded TMR. 24
2.24 Cascaded TMR in an alternative configuration. 25
2.25 Stateful TMR. 26
2.26 DMR checkpointing. 27

3.1 Project overview. 30
3.2 Software tasks. 33
3.3 Software concept 36
3.4 Hardware concept. 39
3.5 Hardware concept with redundant sensors. 39

4.1 First software iteration overview. 43
4.2 Second software iteration overview. 46
4.3 Third software iteration overview. 50
4.4 Final software iteration overview. 55
4.5 Non-redundant hardware. 58
4.6 Redundant hardware. 58

5.1 Test stand for static software tests. 61
5.2 The ground station communication interface. 62
5.3 The ground station sensor data interface. 62
5.4 The standard, non-redundant hardware configuration. 63
5.5 Gyro sensor X-axis output, showing continuous operation. 64
5.6 Gyro sensor Y-axis output, showing continuous operation. 65
5.7 The redundant hardware configuration. 65
5.8 Quaternion test data using task A with motors off. 67
5.9 RPY test data using task A with motors off. 67
5.10 Quaternion test data using task B with motors off. 67
5.11 RPY test data using task B with motors off. 68
5.12 Quaternion test data using task B with motors on. 68
5.13 RPY test data using task B with motors on. 69

List of abbreviations

ADC	Analog-to-digital conversion
BIT	Built-In Testing
MVS	Middle Value Selection
NMR	N-Modular Redundancy
OS	Operating system
SPOF	Single point of failure
TMR	Triple Modular Redundancy
UAV	Unmanned aerial vehicle
WDT	Watch-Dog Timer

1 Introduction

In recent years, the usage of, and research on, unmanned aerial vehicles (UAVs) has seen a steady increase in popularity. While previously primarily a military field, UAVs of all sizes are now becoming increasingly common. As miniturization of components continues, and power demands decrease, the capabilities of such craft also become steadily larger. UAVs can now, besides the aforementioned military uses, be seen in a wide spectrum of uses. Inspecting crops in agriculture, providing previously impossible camera angles for sports events, acting as guides, aiding search operations - the list goes on (NASA [2004], Taub [2013], Sandhana [2013]).

At the Chair of Computer Science VIII at the University of Würzburg, there is a quadrocopter under development called the AQopterI8. An example configuration of this quadrocopter is shown in figure 1.1 below.
The purpose of this research is to provide a fully autonomus quadrocopter capable of flying indoors, where one of the possible applications is to be able to navigate smoke and debris-filled buildings in search of people to be rescued.

Until this point, the AQopterI8 has had limited redundancy in the form of multiple sensors and duplicated sensor readings, offering protection against sensor failures. However, if a central component or software module fails, the entire system will fail, rendering the quadrocopter inoperable. Redundancy for airborne vehicles is always useful, since a failure normally involves the craft crashing. With possible applications such as search and rescue missions, redundancy is even more important.

The goal of this master thesis project is thus to develop a redundancy solution for quadrocopters, allowing the craft to continue operating even in the case of failed components. The redundancy solution will be developed using the real-time operating system RODOS in development at the Chair of Computer Science VIII, in order to investigate its applicability in this field. This document will first describe the current state of the quadrocopter and the RODOS operating



Figure 1.1: The AQopterI8.

system and discuss redundancy solutions that are in use today, followed by a description of the redundancy concept for the AQopterI8. Next, the implementation of the concept will be detailed, and thereafter an evaluation of the implemented solution. At the end of the report, the final section will discuss the obtained results and provide thoughts and ideas for future improvements.

2 State of the art

The state of the art describes the current knowledge base and the foundation upon which the work of this thesis project is built. This consists mainly of three parts:

- The current state of the quadrocopter
- The RODOS operating system
- State-of-the-art redundancy and fault tolerance solutions

First, a description of the current state of the quadrocopter project will be given, which will serve as the initial state for the thesis project. Following this will be a short description of RODOS, the operating system that will be used as a basis for the software to be developed. Finally, current redundancy concepts and implementations will be described in the last section of this chapter.

2.1 Current state of the quadrocopter project

The starting point of this thesis is a fully functional quadrocopter, with a number of hardware and software components. These define the initial state of the project, and serve as a foundation upon which the redundancy functionality will be added. An overview of the initial state of the quadrocopter is shown in figure 2.1 below. As indicated, the three upper boxes show the start-up procedure of the quadrocopter. During hardware initialization, drivers for the sensors, motors and other peripherals are initialized. The following software initialization sets up all the required variables and structures required for operation. The last step of the startup process is calibration, where the sensors as well as the remote control are calibrated.
After the startup process is completed, the program proceeds into the main loop, where it will stay until powered off. The software is thus single-threaded, and as a consequence of this all

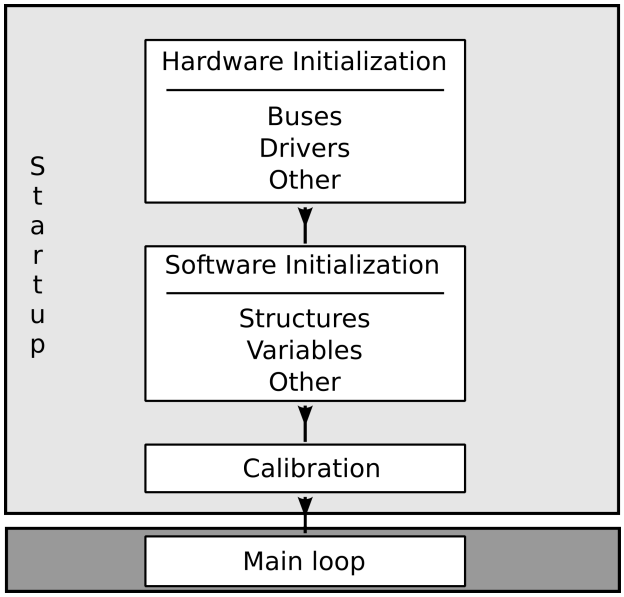


Figure 2.1: Overview of the initial quadrocopter functionality.

sub-processes are run at the same fixed interval. The details of the main loop are shown in figure 2.2.

Each step of the main loop is briefly explained below:

- **Signal processing**

  As already indicated in the figure, the signal processing step is comprised of two parts. The first part is the sensor read, where sensor values are read from the on-board inertial sensors. In the second step, the sensor values are processed into values used for control of the orientation and translation of the quadrocopter. An example of this would be roll, pitch and yaw values.

- **Optional module: Height sensoring & control**

  As the name indicates, in this step the flight altitude is measured and the data is used for height control.

- **Optional modules: Ultrasonic, optical flow & position control**

  The first two steps can be disabled depending on the types of sensors installed. If active, data from ultrasonic sensors is used for collision avoidance, and data from an optical flow sensor is used for maneuvering. Position control, if activated, enables autonomous control of the quadrocopter's position.

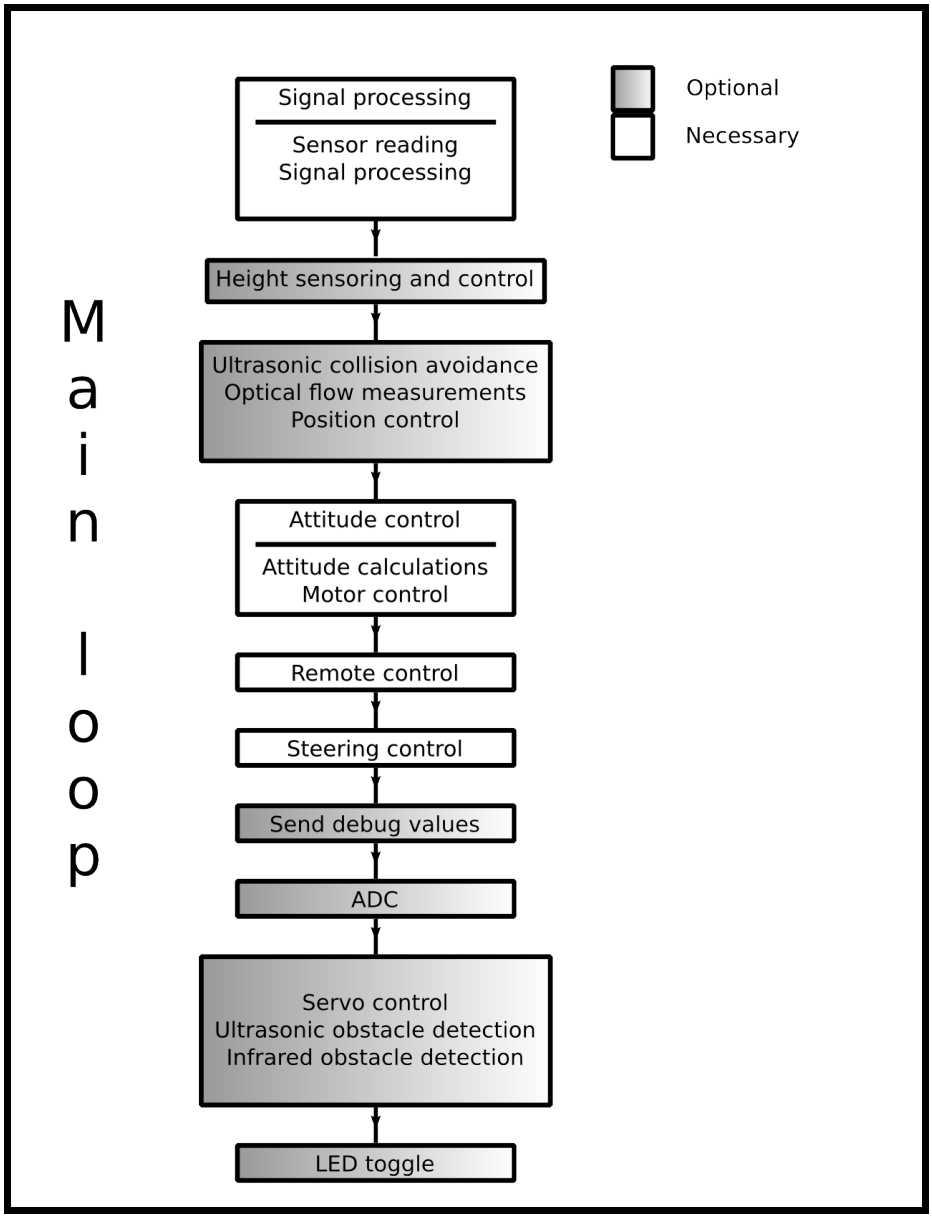


Figure 2.2: Overview of the main loop of the quadrocopter software.

- **Attitude control**

  In this step, sensor data is used to control the attitude of the UAV, including transmission of motor commands to the motors to maintain the desired attitude.

- **Remote control**

  In this step, commands from a remote control are read and parsed for usage in the quadrocopter.

- **Steering control**

  Here, the actual method of control is decided. Examples are full autonomous flight, remote control, height control, and more.

- **Optional module: Send debug values**

  As the name implies, debug values are here sent to the ground station.

- **Optional module: ADC**

  Analog-to-digital conversion (ADC) occurs here.

- **Optional modules: Servo control, ultrasonic and infrared obstacle detection**

  In this optional step, depending on the installed hardware a combination of the following tasks is performed: servo control depending on attitude to keep the platform level; obstacle detection using either ultrasonic and/or infrared sensors.

- **Optional module: LED toggle**

  As the name implies, here an LED is toggled. This is used as a visual aid to see that the loop is running.

Before proceeding to the next section, it is also good to examine the data structure of the initial version of the software. The software is running on bare metal in a single thread, and data is communicated in the form of global variables. Different modules use the same global variables for their input and output. This is illustrated in figure 2.3.
This concludes the overview of the initial state of the quadrocopter software.

2.2 The RODOS operating system

RODOS is a real-time, multi-thread-capable operating system initially developed by the German Aerospace Center and it is currently being developed in cooperation with the Informatics VIII Chair at the University of Würzburg (DLR [2013], JMUW [2012]). The system is based on a set of fundamental ideas, including the following:

- Dependability

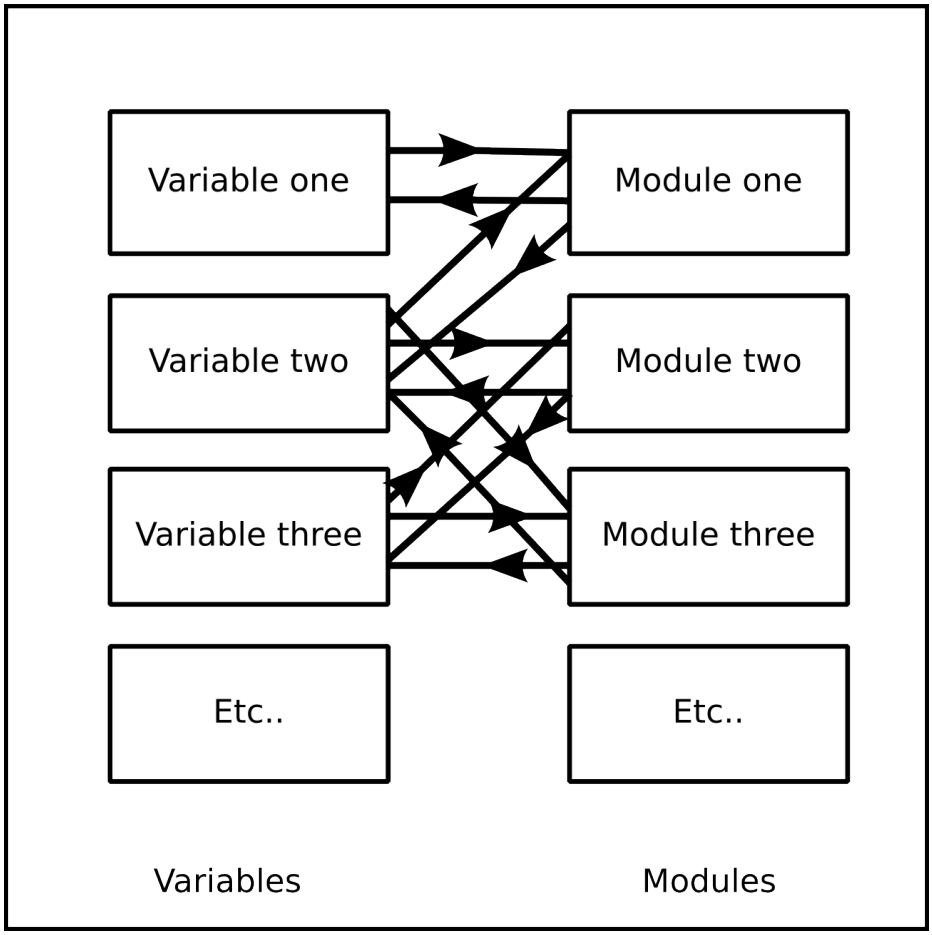


Figure 2.3: Overview of the data structure and inputs/outputs of the quadrocopter software modules.

- Simplicity
- Modularity

The items in the list are all connected, as will be shown below. The basic principle behind the operating system is to offer a solution which is as simple as possible, but not simpler - this is the idea of irreducible complexity (Montenegro [2008]). Simplicity in turn leads to greater dependability, since increasing complexity brings increasing risk of failure due to the more involved interactions between a higher number of components in a more complex system. However, while remaining as simple as possible, RODOS still provides all the necessary functionality that is required of a real-time operating system (Montenegro [2008]).

The third item in the list describes how software applications are developed for RODOS. Using the systems services, especially the middleware layer, applications can be developed as indepen-

dent, interchangeable modules with run-time reconfiguration capabilities, and how this is done is explained below.

Additionally, in the scope of this project, one of the more important characteristics of RODOS is that it supports multi-threading. In order to provide a better understanding of such a configuration, first a few basic principles of multi-threaded systems are described.

2.2.1 Operating systems and multi-threading

To provide a clear foundation for the principles of multi-threading described below, the scope will be limited to the type of system that is used in this project. The basis of this project is a single-core processor in an embedded system, and it is for such a system that the following principles are valid. Modern processors, such as the ones found in most personal computers, may have multiple cores or architectures that in other ways enable concurrent execution, and the description below is not valid for such systems.

In the scope of this project, a major advantage of using RODOS or a similar operating system is thus the possibility of multi-threading. By running several so-called *threads*, the applications can run virtually simultaneously. This concept is shown in figure 2.4.

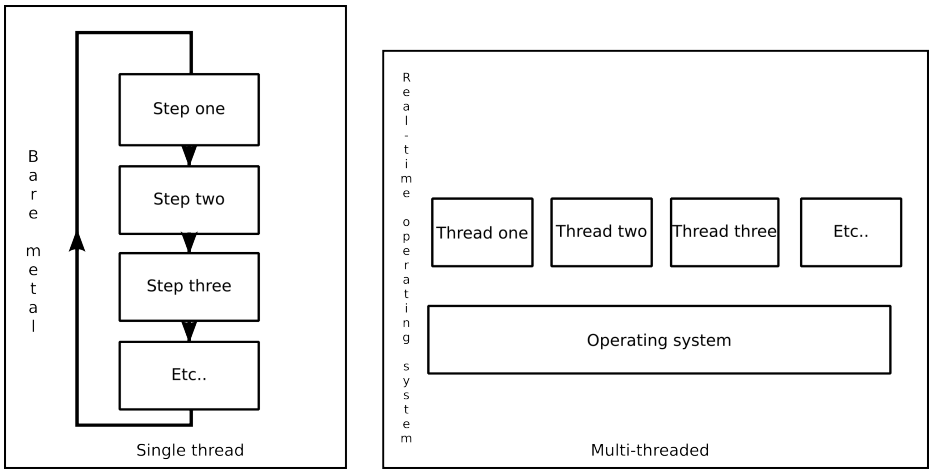


Figure 2.4: Comparison between single-threaded and multi-threaded designs.

The assumption of simultaneous execution as shown in the illustration is only valid at a macro scale. Only one thread can run at any one time, and CPU time is allotted to each thread by the scheduler of the operating system depending on thread priority. An example of such time allocation for different threads is shown in figure 2.5.

Despite not being entirely accurate, the idea of simultaneous execution is useful as a concept. It

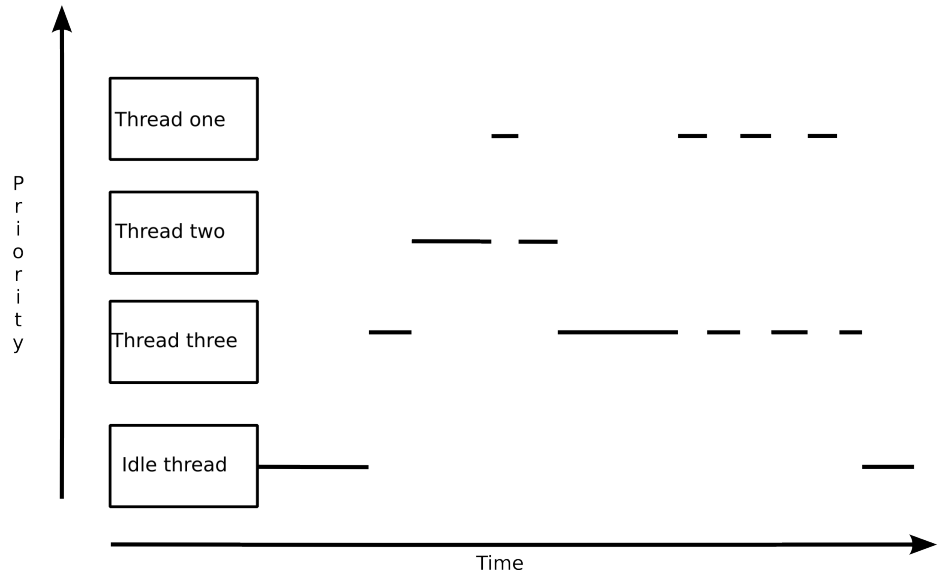


Figure 2.5: Example of thread scheduling.

does, however, put certain constraints on the data structure and data access in a program. These will be discussed below, starting with data access.

2.2.2 Data access

With a multi-threaded system, special considerations have to be taken regarding how data is stored and accessed. This can best be illustrated by a comparison with single-threaded software. Consider the following figure, which is a partial close-up of figure 2.3.

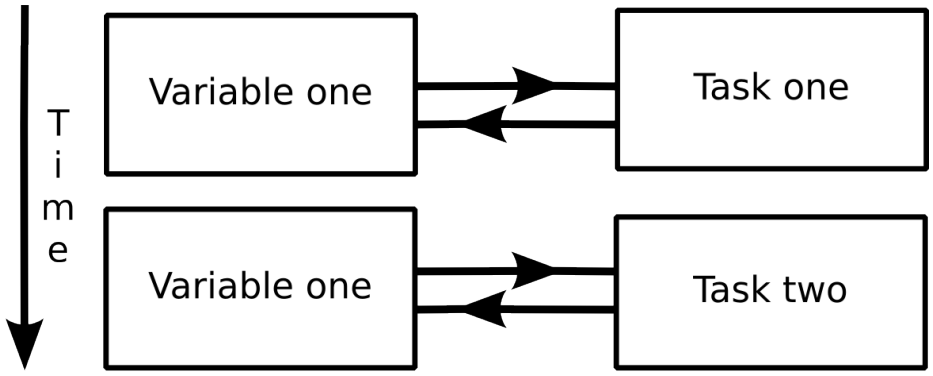


Figure 2.6: Single-threaded data access.

In a single-threaded system, such a setup is perfectly viable. Task one first reads from and writes to the variable, followed by read/write operations in task two. All operations are sequential, and there is no risk of any corruption or conflicts. Now consider the same situation in a multi-threaded

environment, where read and write operations for different tasks are not necessarily strictly sequential. Depending on what variables are being accessed, unchecked variable access in such a system can have consequences ranging from incorrect data being used to memory corruption and system failure. The basic premise of this problem is simple, and is illustrated in figure 2.7.

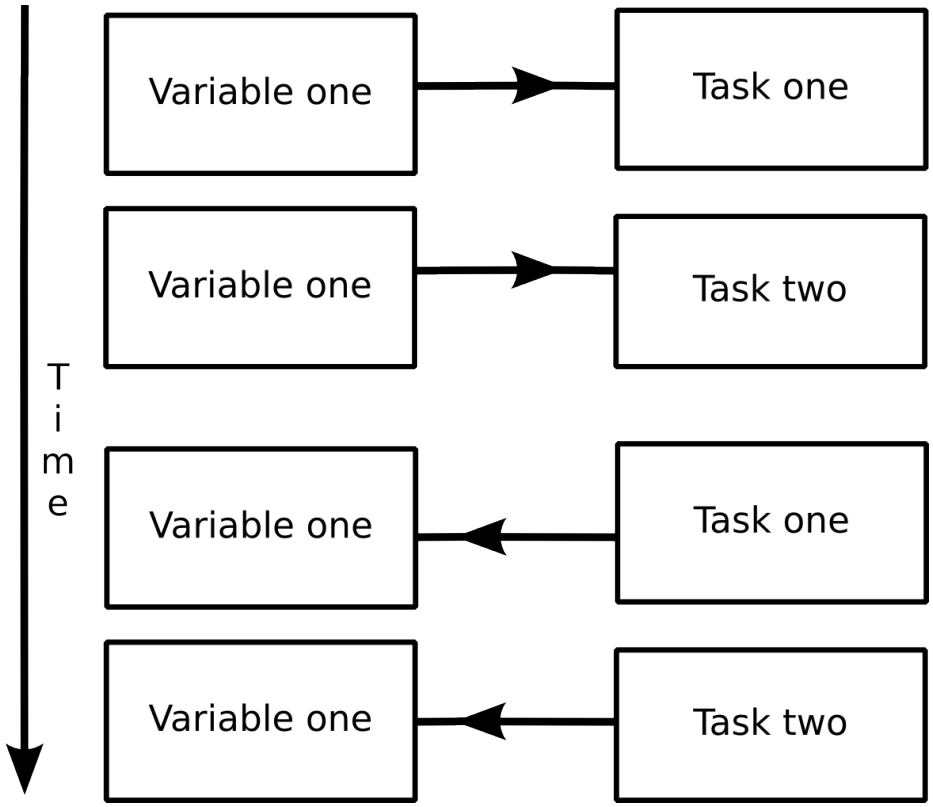


Figure 2.7: Multi-threaded data access.

Here one should remember that threads do not actually run simultaneously, but are rather allotted time by the system scheduler. In the figure, the first task reads data from the variable in order to process it. In a single-threaded system, the variable would then have been updated, and its value would have been used for calculation in the second task, as is already shown in figure 2.6. However, in this system, the second task is activated before the first task finishes. Now, the value that is read from the variable has not yet been updated by the first task and may result in incorrect or outdated output from the second task. A good example of this is a very basic operation in programming: incrementing a counter. If both tasks are set to increment a counter, in this case our variable, both tasks would read and return the same values. The end result would be a missing counter increment. One could easily imagine more severe consequences, depending on the type of variable that is being read. In the case of pointers, invalid memory access and memory

corruption are possible outcomes. Since multi-threaded systems are commonplace, there are of course a number of ways to solve such issues. In this project, the two following methods will be used:

- Mutual exclusion
- Local storage

Mutual exclusion is a method which locks data to one task, making sure that any access to shared variables can only be performed by one thread at a time. Variable access is thus serialized, and the problem of multiple threads accessing the same data is thus avoided. However, if care is not taken while using this method, it can spawn problems of its own. A serious example of this is what is known as a deadlock, where two or more threads are depending on shared data that is currently locked by the other thread(s). The program thus cannot proceed, as none of the threads can continue execution without access to data which is locked. This method is thus employed sparingly and with care taken to avoid this type of dependencies.

The second method, local storage, is a major part of the concept. This method removes the shared data by introducing variables that are local to each thread, thus bypassing the problem. Although such a solution simplifies multi-threading, it also has its disadvantages. Thread local variables must still be updated with data from other threads, requiring inter-thread communication and means to buffer such communications. Another disadvantage is that since each thread will have local copies of the data plus any additional buffers required, memory usage is increased.

Since local storage relates to the data structures to be used, the concept for such storage and the accompanying inter-thread communication will be described in the following subsection.

2.2.3 Data structure

In order to use local storage, a way of communicating between threads must exist. In this project, such communications will use the middleware layer of the operating system, which is based on a publisher-subscriber protocol (Montenegro [2008]). An illustration of this scheme can be seen in figure 2.8

The publishers can be, for instance, sensors and software modules that provide new data; subscribers are units which consume data, such as actuators or software modules requiring inputs. As is shown, subscribers and publishers communicate via topics, which function as an interface.

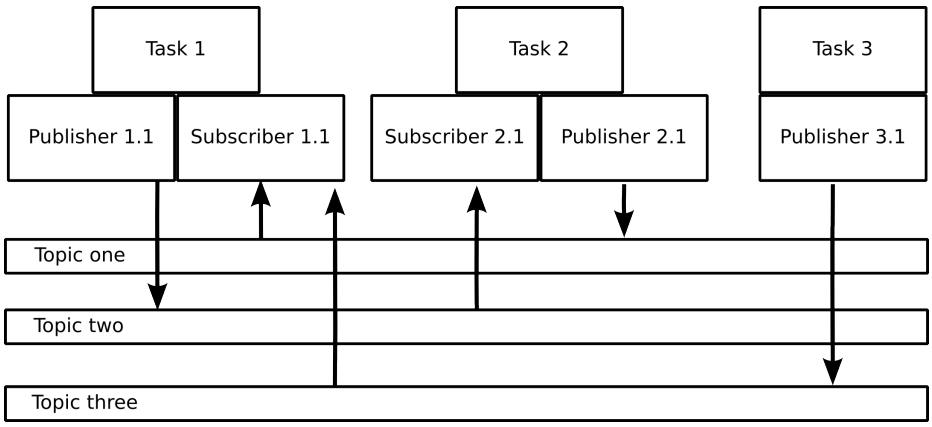


Figure 2.8: Inter-thread communications.

Topics, which consist of an identifier and a data type, can be subscribed to by subscribers. Publishers, in turn, use a topic to publish new data to all subscribers which are subscribed to that topic. This allows for much flexibility and modularity when designing the software. Using topics and a publisher-subscriber protocol also confers additional advantages: software and hardware components are interchangeable, with all communications being handled by the middleware, and the system can thus also be reconfigured at run-time (Montenegro [2008]).

2.3 Hardware redundancy solutions

Fault tolerance and reliability are important parts of any system where a loss of function or a malfunction is deemed too costly to be allowed. Although redundancy can be found in virtually every field, the context of this thesis is electronics in general and avionics in particular. Therefore, that is were the focus of the next three sections will be. This section and the following describe general strategies for redundancy in hardware and software, respectively. In the last section of this chapter, a number of specific examples of redundancy solutions are shown.

In order to clearly show the different strategies that can be employed in order to provide fault-tolerance in hardware, it is useful to first provide an example of an initial system architecture which is not fault-tolerant. The simplest way to describe such a system is that it receives inputs, processes them and then produces outputs. Such an example is shown in figure 2.9.

As can be seen, if any one component or connection in the system fails, the entire system will stop functioning. In many less critical applications, such an architecture is acceptable (and economically even preferred, due to the increased cost of making the system more reliable) as long



Figure 2.9: System without fault-tolerance.

as the cost of a system failure is sufficiently low. However, this type of vulnerability which is known as a single point of failure or SPOF, is to be avoided for systems where a high degree of reliability is desired; thus the need for redundancy and various strategies to improve reliability and decrease the risks of total system failure.

Before moving to redundancy, however, an important distinction should be made between failure types: *loss of function* and *malfunction* (Hammett [2001]). A loss of function means that the affected component stops working and sends no data; a malfunction means that the component behaves erratically and produces invalid data. This distinction is good to keep in mind while reading the following section, as the various fault-tolerance strategies described below have different capabilities of handling the two failure types.

2.3.1 Built-In Testing

A simple way to give a moderate increase in reliability to a system is by including some form of built-in testing, or BIT. A widely used form of BIT is that of a watchdog timer, or WDT. The concept for such a component is simple. The watchdog timer is set to a specific interval, after which it "times out". Depending on the configuration of the system, a timeout often causes a reset of the unit which the WDT is assigned to, and it can also notify other parts of the system about the occurrence of a watchdog reset. In order to avoid a reset, the unit which is under supervision of the WDT must periodically send a command which resets the timer. Failure to do so would result in a reset. Thus, a WDT protects a processor from getting stuck in loops or otherwise becoming unresponsive. An example of this setup is shown in figure 2.10 below.

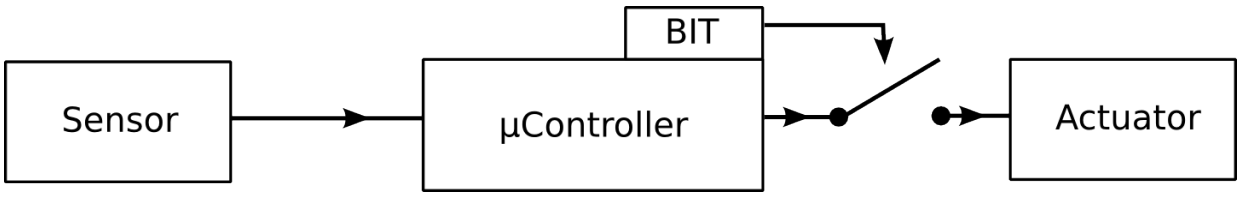


Figure 2.10: System with built-in testing.

It can easily be seen that this strategy does not provide any protection against failures which lead to a loss of function, as there is still only one channel for the data. However, in case of a malfunction, where the processor produces erroneus data, the built-in testing component can detect this and disconnect the processor from the output. Thus, built-in testing provides a way to protect a system against malfunction (Hammett [2001]).

2.3.2 Dual redundancy

This strategy involves the use of an additional processing unit, as is shown in figure 2.11.

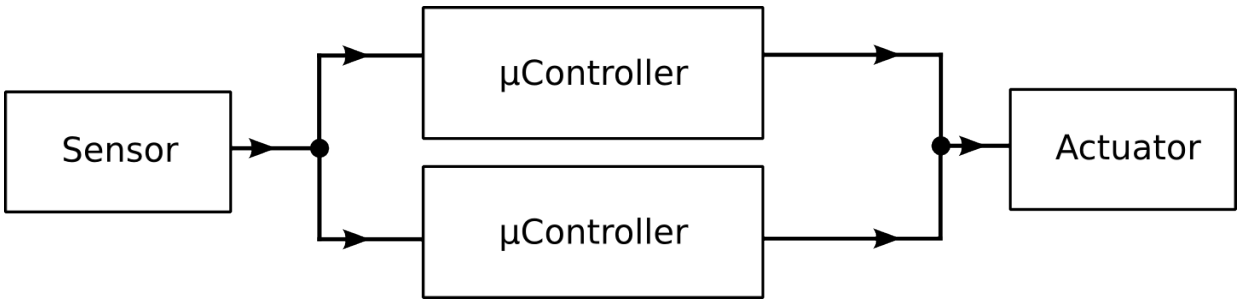


Figure 2.11: System with dual redundancy.

The shown configuration allows one of the processing units to fail, while leaving the system operational. Depending on the needs of the system, the redundant processor can be run in either a so-called *hot redundancy* or *cold redundancy* mode (Yellowley [2006]). In case of hot redundancy, any state information stored by the primary unit is shared with the backup unit. In case of cold redundancy, this is not the case, and the backup unit starts with no information about the state of the system. Therefore, hot redundancy works faster when a unit fails, but creates a certain overhead due to the data updates for the backup. Cold redundancy, on the other hand, works slower but gives a less complicated system, while also not creating any additional overhead. Thus, depending on the required maximum failover time following a primary processor failure, the mode of redundancy is selected (Hammett [2001]). In the context of a quadrocopter, the required failover time is at most a few hundred milliseconds - using cold redundancy, where the backup unit needs to perform a full startup sequence following a failure of the primary processor, is therefore not possible and hot redundancy is required. In order to utilize the redundant unit, there must be a way to detect a failure of the primary processor. This can be done in a number of ways; a common way is to combine it with a WDT, yielding control to the backup processor if the primary does not reset the timer within the given interval (Kim und Kim [2011]). This is

shown in figure 2.12

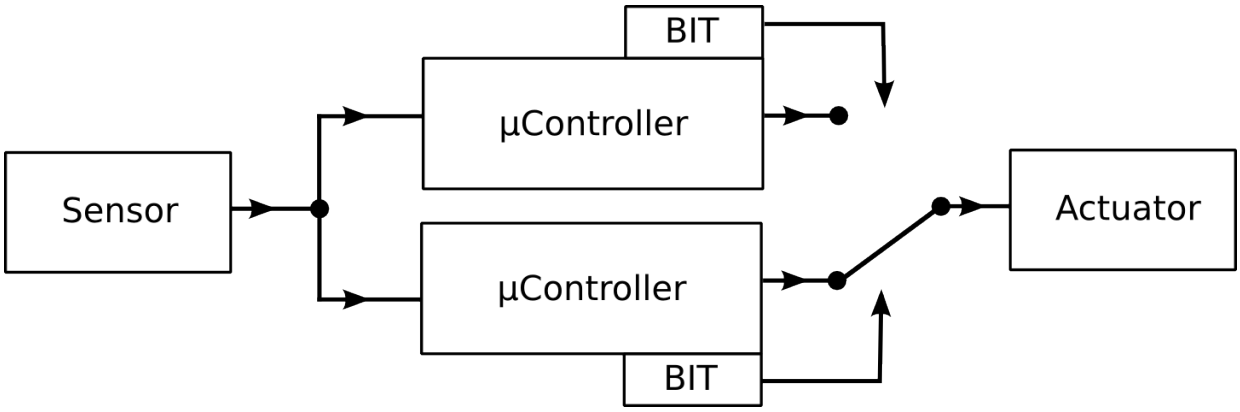


Figure 2.12: System with dual redundancy and BIT failover.

Using two redundant processing units, other architectures are also possible (Son et al. [2007]). The solution shown above provides protection against a complete loss of function in the event of a processor failure; however, there is no way to determine if the primary processor malfunctions (Yellowley [2006]). A method of detecting such malfunctions is to arrange the two processors in what is known as a *self-checking pair* (Bolduc [2001]). This architecture is shown in figure 2.13.

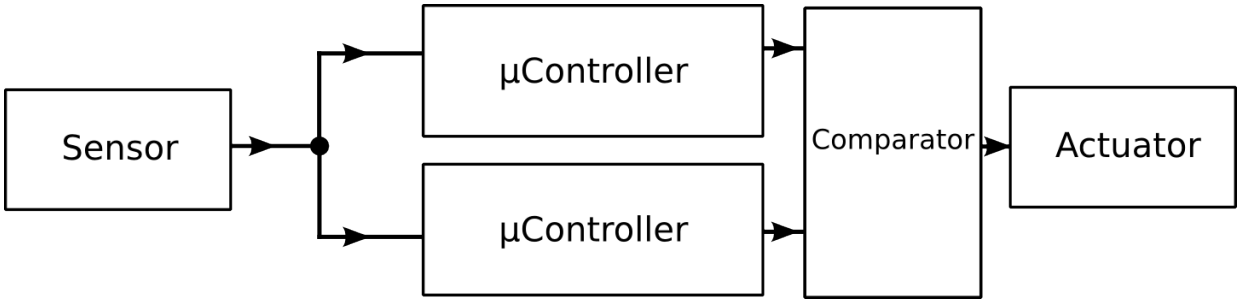


Figure 2.13: System with dual redundancy as a self-checking pair.

If any of the processing unit fail, the failure is detected when their outputs are compared - only matching outputs are forwarded, and the error thus cannot propagate to the actuators. However, a consequence of this setup is that a failure in any of the processors leads to a failure and loss of function of the entire system. Thus, even though there are two processors, the risk of a loss of function is actually greater than in a system with no redundancy. This owes to the fact that both processors are required for operation, and their failure probabilities are compounded when examining the full system. Because of this, a self-checking pair in the configuration shown is not a good solution when a loss of function is unacceptable (Hammett [2001]), and only provides pro-

tection against malfunction. A way of modifying the architecture to reduce the risk of losing the entire system requires the introduction of a new concept - *fault-down*. The concept of fault-down involves the reconfiguration of a redundant architecture following a failure in any of its redundant units, allowing the system to remain functional while reducing the level of protection against loss of function or malfunction. An example of this concept is shown in figure 2.14.

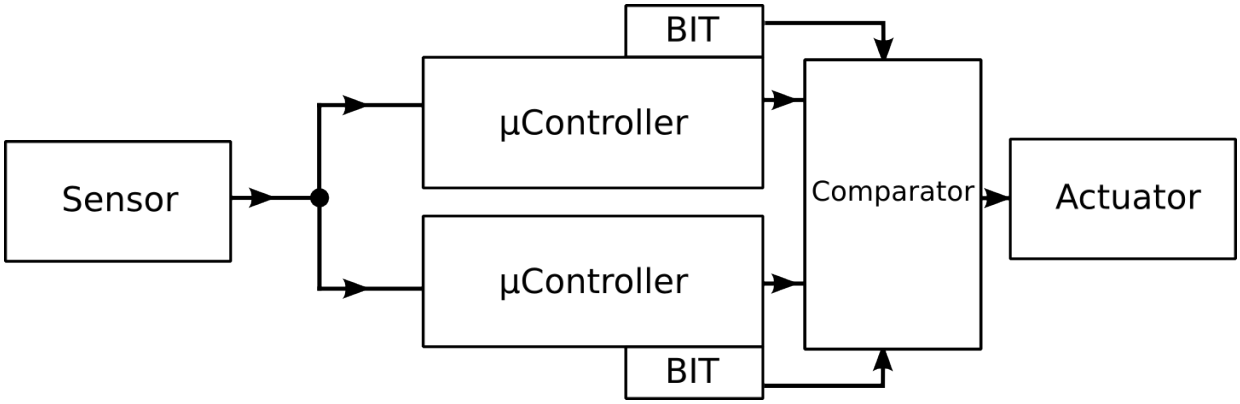


Figure 2.14: Self-checking pair with fault-down capability.

Hardware-wise, the difference is small. The BIT components are added back, having previously lacked a function in a basic self-checking pair. Their inclusion now is linked to a change of the logic in the component which compares the output of the two processors (Chandhrasekaran und Choi [2010], Kim und Kim [2011]). In a basic setup, output is only passed on to the actuators if both outputs match, preventing any malfunction while increasing the risk of a loss of function. In this modified approach, the signal can pass through even if there is no agreement, providing that one of the BIT components indicate that a failure has occurred (Bolchini et al. [2002]). In effect, the system then becomes a single-channel system with no redundancy, while avoiding a complete loss of function. This reconfiguration, as was mentioned earlier, is known as a fault-down.

2.3.3 Triple redundancy

The difference between malfunction and loss of function has been described above, as well as different architectures that can protect against either of these fault conditions. However, none of the architectures presented can adequately protect against both. The one that comes closest is the self-checking pair; however, it can only detect a malfunction and there is no guarantee that it stops the error from propagating, since it lacks any redundancy in its fault-down mode. In order to protect against both loss of function and malfunction, additional hardware is required - hence

the need for triple redundancy. One of the more widely used forms of redundancy is the so-called *triple modular redundancy*, TMR (Aysan et al. [2008], Bolduc [2001], Hamamatsu et al. [2010]). An example of this architecture is shown in figure 2.15.

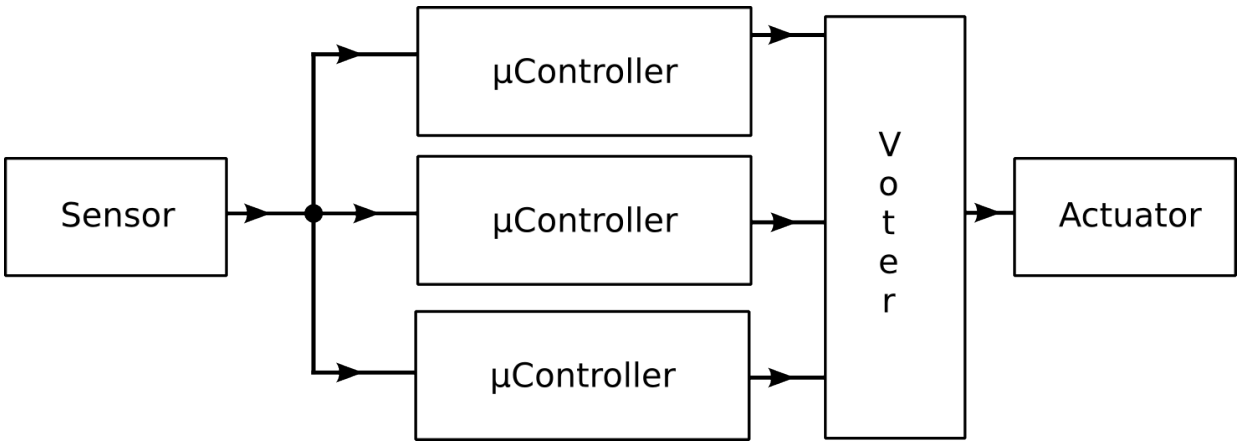


Figure 2.15: Triple modular redundancy

The idea behind TMR is straightforward. Using three processing units, following a single failure in a processing unit there will always be a majority of healthy units that can provide the correct output to send to the actuators. There are different strategies, each with different strengths and weaknesses, that can be employed in order to mask out any faulty output. Two common strategies are majority voting (Hammett [2002]) and middle value selection, or MVS (Ahlstrom et al. [2002]). When using majority voting, the voter selects the pair of computers whose outputs match. This can be done using either an exact or approximate match, where the exact method puts higher demands on synchronization (Bolduc [2002]) but also gives a higher potential for fault detection. Majority voting can also cause problems in situations where none of the outputs match, and special attention has to be paid to such situations when designing the system. An advantage of majority voting is that it enables the system to recognize that an error has occured and to find the faulty component in order to take appropriate action, such as isolating the unit.

MVS does not use voting of the different values; instead, the voter simply selects the value that is between the other two values. A faulty unit is assumed to produce a value which is much different from the values of the healthy units, given the same inputs, and its output would thus not be selected. Since this strategy does not clearly label any unit as faulty, fault detection is more difficult using this method compared to majority voting. However, there is less demand for synchronization of inputs/outputs and state data. Faulty units can also be detected by the use

of threshold values. MVS thus provides a simpler way with the disadvantage that detection and isolation of a faulty component is more difficult than in the case of majority voting.

A basic requirement of TMR that should be noted and explained is that of voter reliability. As was already mentioned above, the idea of redundancy is to avoid any SPOF. Despite this requirement, the voter itself represents a SPOF, regardless of the redundancy of the multiple processing units. In order to be effective as a redundancy strategy, the constraint set upon the voter is that it should be more reliable than the processing units whose outputs it selects between. This can be accomplished in different ways; simpler components such as logic gates (Hamamatsu et al. [2010]), or simpler programs in case of software implementations.

2.3.4 Redundancy requiring more than three redundant units

There are additional architectures that can be used which require more than three redundant units (Gostelow [2011], Stojcsics [2012]). Indeed, depending on the number of faults that must be tolerated, the previously described architectures can be scaled up according to demand. However, there is also the factor of diminishing returns - once the number of redundant units reaches a certain level, the increased complexity and thus higher risk of failure of the redundancy management system itself may offset the increase in reliability gained by the additional redundant units.

An extension of the TMR concept described above is the N-modular redundancy (Aysan et al. [2008], Xiaojun et al. [2012]), or NMR- alternatively, TMR can be seen as a special case of the more general NMR concept. N here stands for the number of channels (redundant units) in the architecture, and TMR can thus also be described as 3-channel NMR. 4-channel NMR is used in a number of applications where reliability must be exceptionally high, such as airliners and manned spaceflight (Racine et al. [2002]). Another 4-channel architecture is based on the self-checking pair as described in section 2.3.2, negating the disadvantage of an increased risk of loss of function. In such a setup, the self-checking pair is doubled, thus allowing for a redundant pair in case the output of the primary pair does not match (Yellowley [2006]). Examples of the 4-channel NMR and dual self-checking pair architectures are shown in figures 2.16 and 2.17, respectively.

This concludes the section on general hardware redundancy principles. In section 2.5, examples

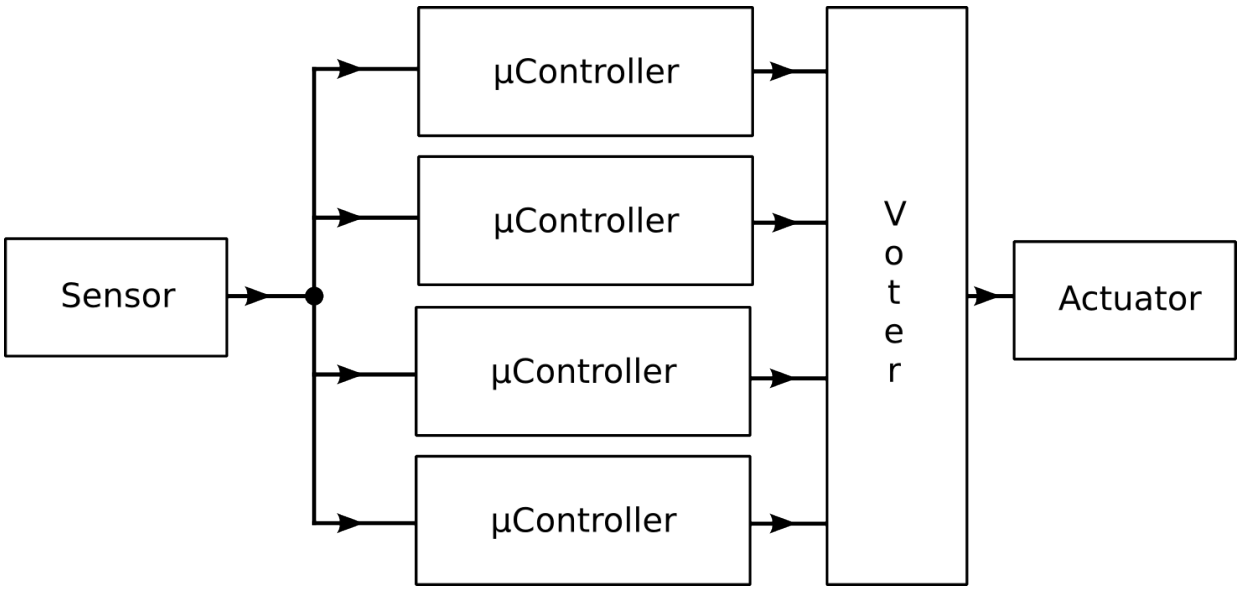


Figure 2.16: 4-channel N-modular redundancy.

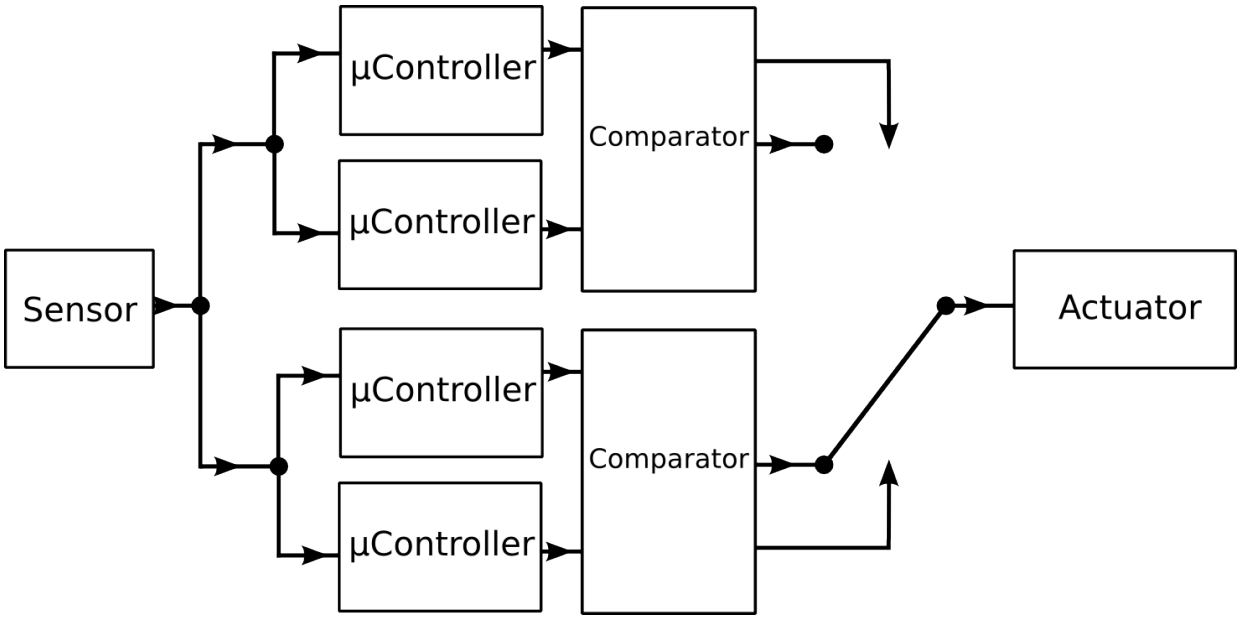


Figure 2.17: Dual self-checking pair.

of state-of-the-art implementations of the described architectures will be discussed, together with any modifications and extensions of the basic concepts.

2.4 Software

In the case of software fault tolerance, there are a number of different techniques that can be employed. Two common techniques are temporal redundancy and software diversity, which both will be discussed below, starting with temporal redundancy.

2.4.1 Temporal redundancy

The concept of temporal redundancy, or time redundancy as it is also called, is fairly simple. In order to prevent any temporary disturbances (so-called *transient faults*) from affecting the system, systems with temporal redundancy provide a means to repeat program execution. A common implementation of this scheme is in the form of checkpointing (Ayav et al. [2008]), where data is stored at regular intervals. If any process produces erroneous output, or is otherwise not properly executed, that process can be restarted using the stored data as input, thereby avoiding any transient faults. This concept is shown in fig 2.18.
The advantage of such a setup is that transient errors can be prevented from producing faulty system output by repeating the calculation. Disadvantages include increased processing time and memory requirements, since data has to be stored for each cycle. Depending on the type of checkpointing, other disadvantages may also apply. If the data is compared in order to find if an error has occurred, a permanent fault in one process may lead to an infinite rollback as the outputs from the healthy and faulty processes will never match. This has to be considered when designing the system, leading to increased complexity.

2.4.2 Software diversity

The concept of software diversity is used to avoid that any one error type will cause a malfunction or loss of function. The basic idea is based on two or more different version of software running in the system, producing identical inputs and outputs. In case any input, or other cause of error, is introduced, the risk of it affecting two dissimilar programs is less than if all software is identical (Janarthanan und Gherbi [2011]). This type of redundancy can either be used in a single computing unit, or in combination with hardware redundancy, using different software versions in different redundant hardware units. These two modes of software diversity are illustrated in figures 2.19 and 2.20, respectively.

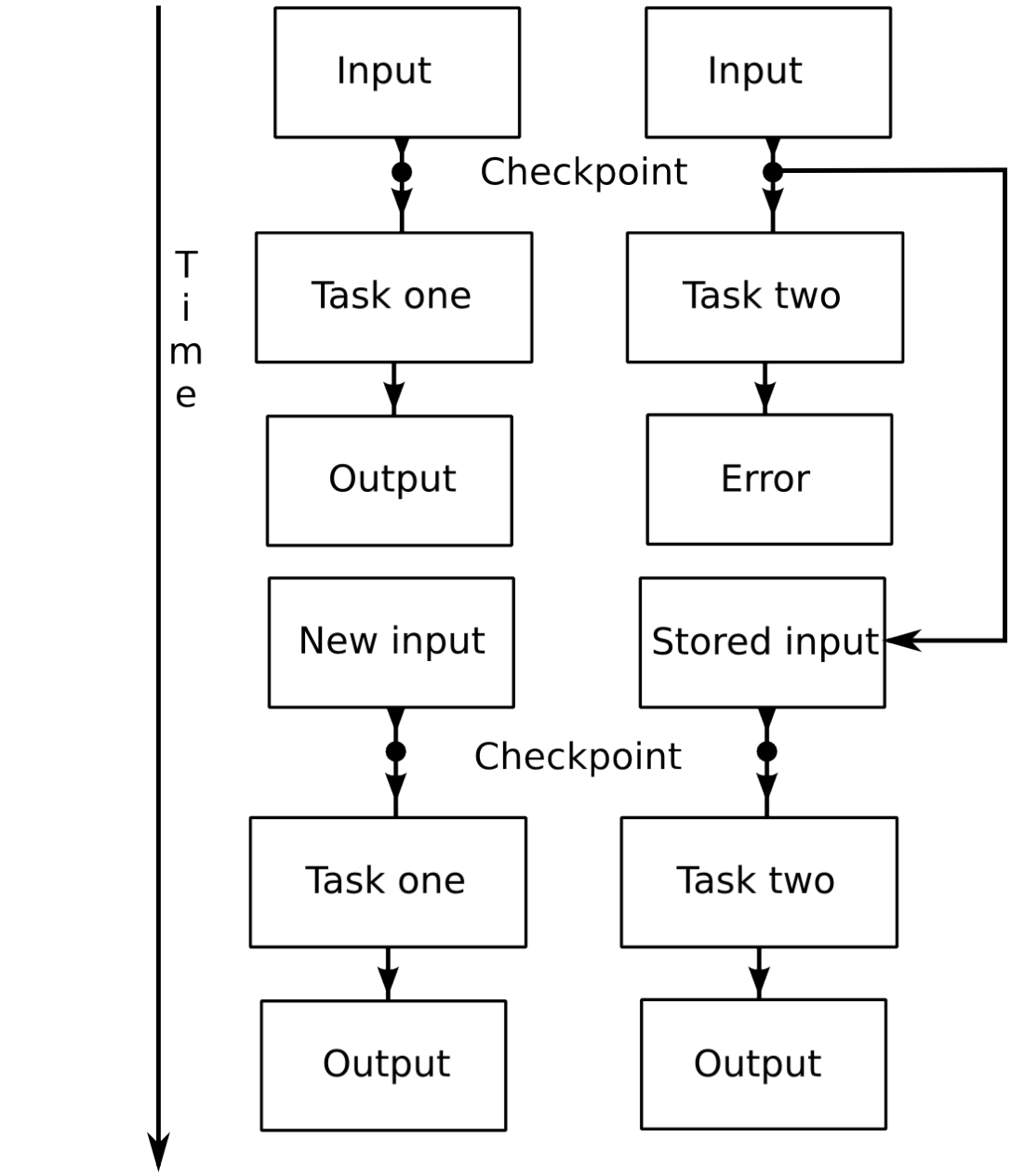


Figure 2.18: Checkpointing and rollback.

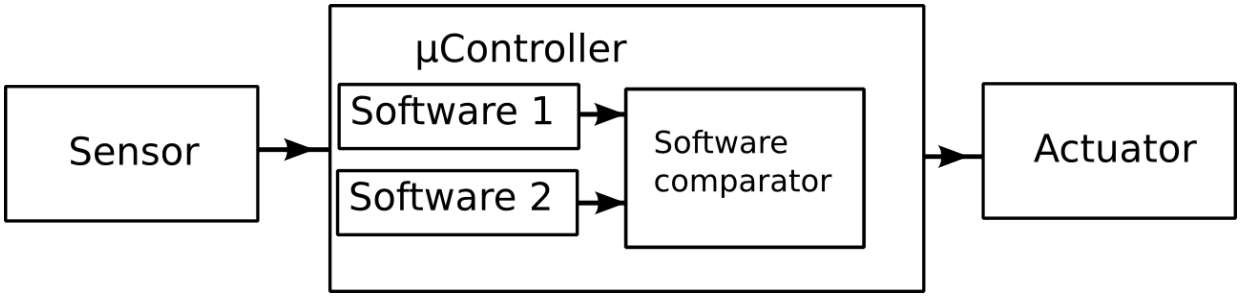


Figure 2.19: Software diversity in a single hardware unit.

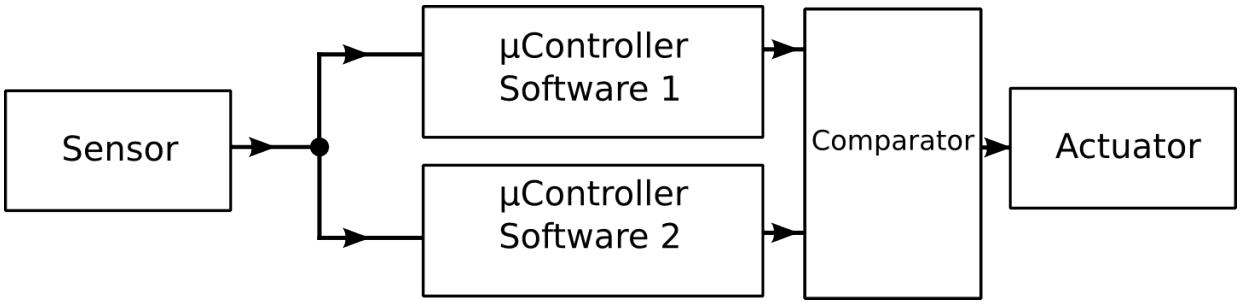


Figure 2.20: Software diversity in multiple hardware units.

Another kind of software redundancy that can be used is a software analogue of an already described technique for hardware redundancy; TMR. In this case, instead of using separate hardware units for each channel, all redundant channels and the voting are executed in the same physical node (Ulbrich et al. [2012]). This setup is illustrated in figure 2.21.

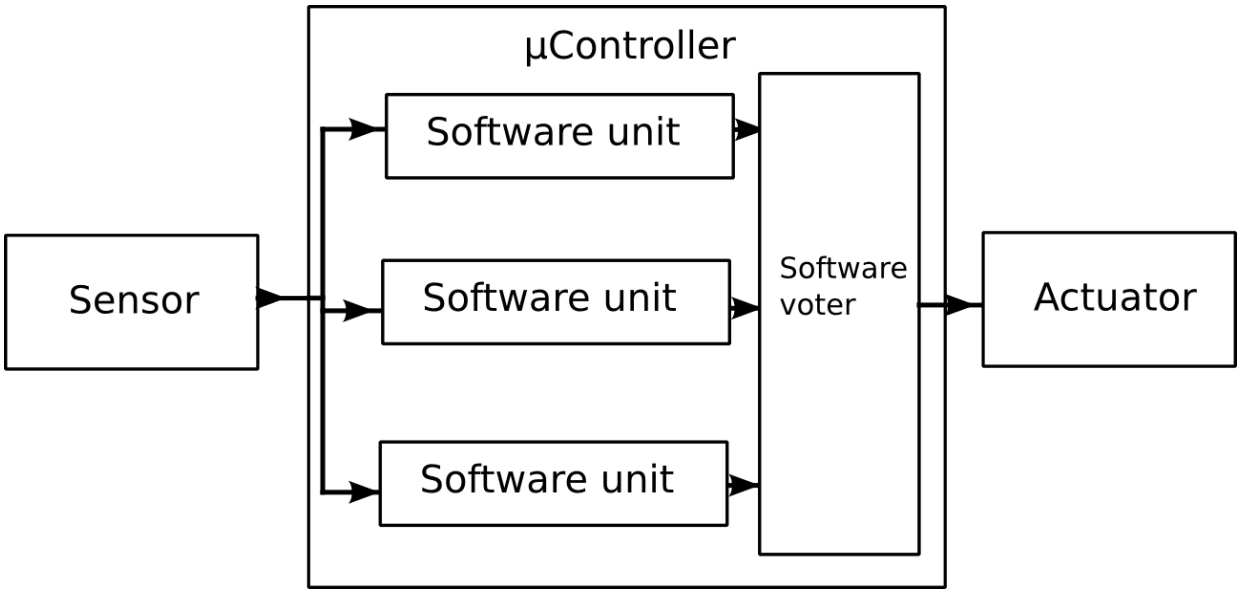


Figure 2.21: Software TMR.

The concept does not protect against loss of function, as all computations are still executed in one physical unit. It does however protect against any transient faults, as any faulty output due to such a fault will be discarded in the voting process. It is also possible to use both software and hardware TMR in a system, thereby increasing the level of redundancy and fault handling. An example of such a setup is shown in figure 2.22

This concludes the sections on basic redundancy principles. In the following section, a number

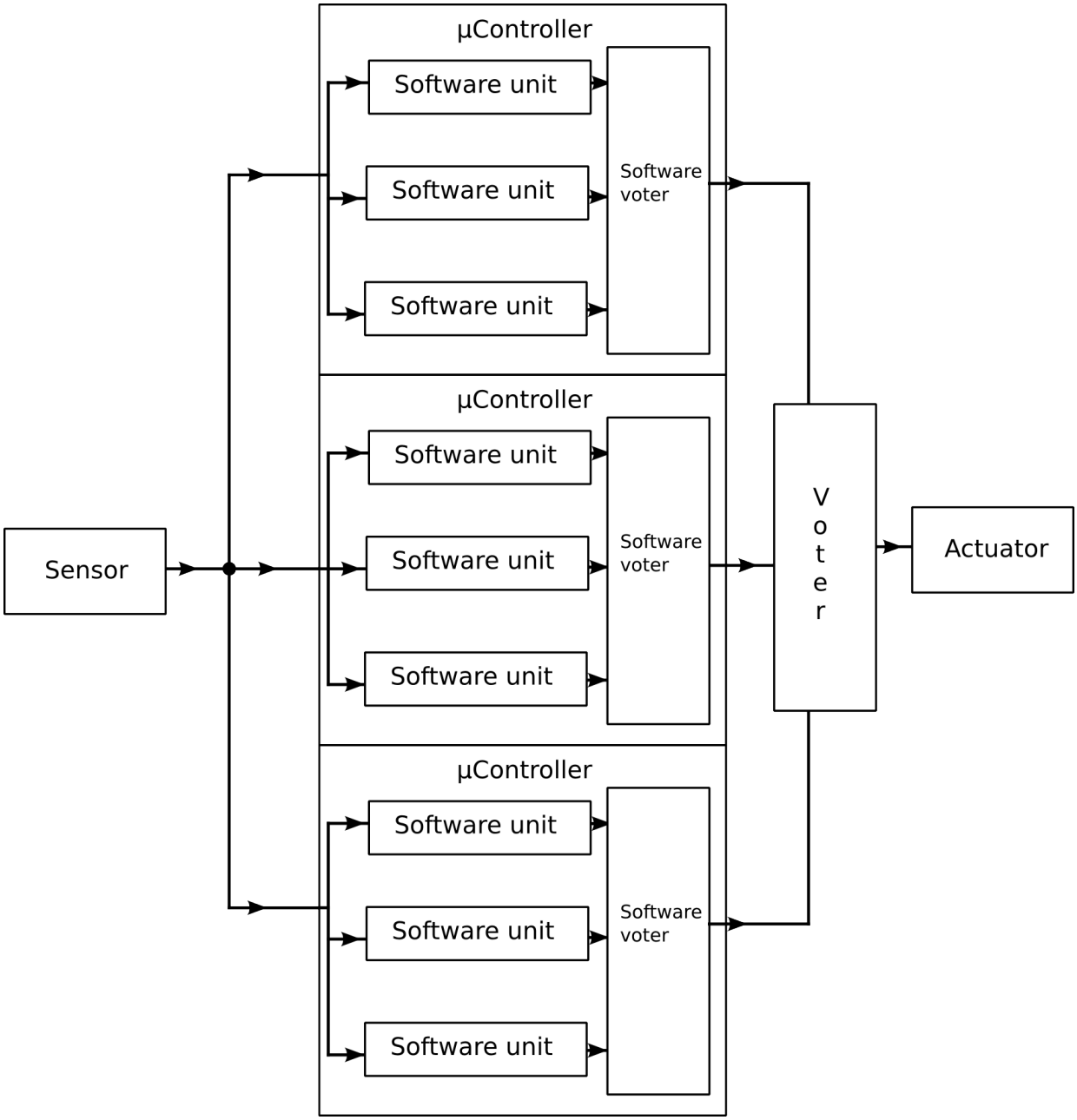


Figure 2.22: Software and hardware TMR.

of examples are shown where these techniques are used, in various forms, versions and combinations.

2.5 Examples of fault tolerant systems

Due to its popularity, the first examples to be shown are based on TMR. Following those, the next examples will show various version of temproal redundancy, in combination with hardware redundancy.

2.5.1 Cascaded TMR

Cascaded TMR uses the initial concept of TMR by adding a number of such configurations in series. Please refer to figure 2.23 for an illustration.

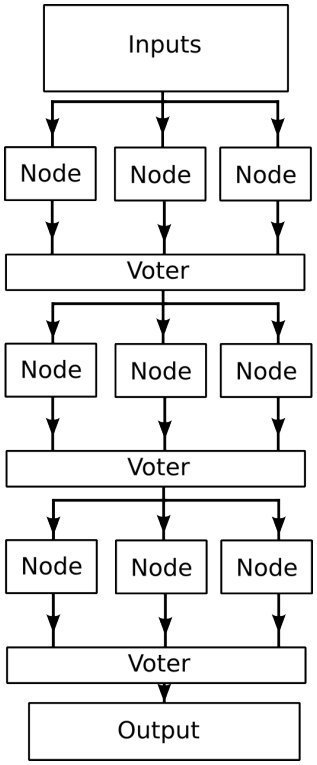


Figure 2.23: Cascaded TMR.

Such configurations have been used, although they have a drawback in the fact that there are multiple SPOFs, namely the voters. If one voting unit fails, the entire system fails. Therefore, voter reliability must be kept high in order to use such a setup. In recent developments, alternative configurations have been used (Hamamatsu et al. [2010]), which bypass the problem of voter failure. This is illustrated in figure 2.24.

Any voter failure in such a configuration is handled in the subsequent stage, thereby eliminating any SPOF and increasing reliability.

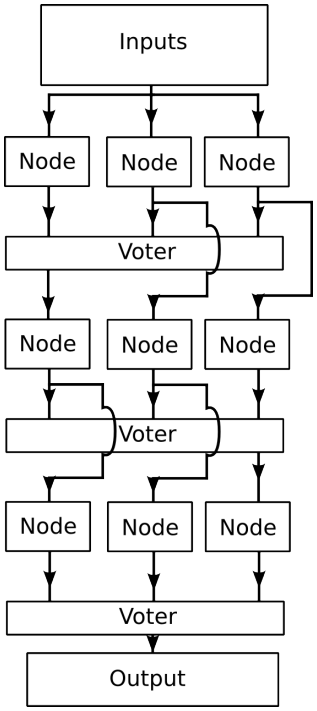


Figure 2.24: Cascaded TMR in an alternative configuration.

2.5.2 Stateful TMR

Stateful TMR is another method of improving the reliability of standard TMR in the case of binary outputs (Matsumoto et al. [2010]). In addition to taking into consideration the outputs of each node, any previous failures are also taken into account. An illustration of this technique is provided in figure 2.25.

While normal TMR would fail and produce faulty output after two nodes have failed, Stateful TMR still provides a correct output. Since the node that first fails is set as unreliable in the 2nd iteration, when the second node fails and the faulty nodes thus gain a majority, the system can recognize that the majority is faulty and the system thus does not fail. Only after a third and final fault does the output become faulty. However, this is only true for situations where one node fails at a time. When two nodes can fail simultaneously, Stateful TMR can instead be less reliable when compared to a standard TMR solution (Matsumoto et al. [2010]). If first one node fails, it is marked as unreliable. If that node then recovers, but another node fails, the previously faulty node will be in majority. Since it is marked as unreliable, the Stateful TMR solution will see the majority vote as unreliable also, and select the faulty node for output. A standard TMR solution would have selected the two healthy nodes instead. Thus, Stateful TMR improves reliability when the risk of a node failure is low, but decreases reliability when the risk is high. In order to address

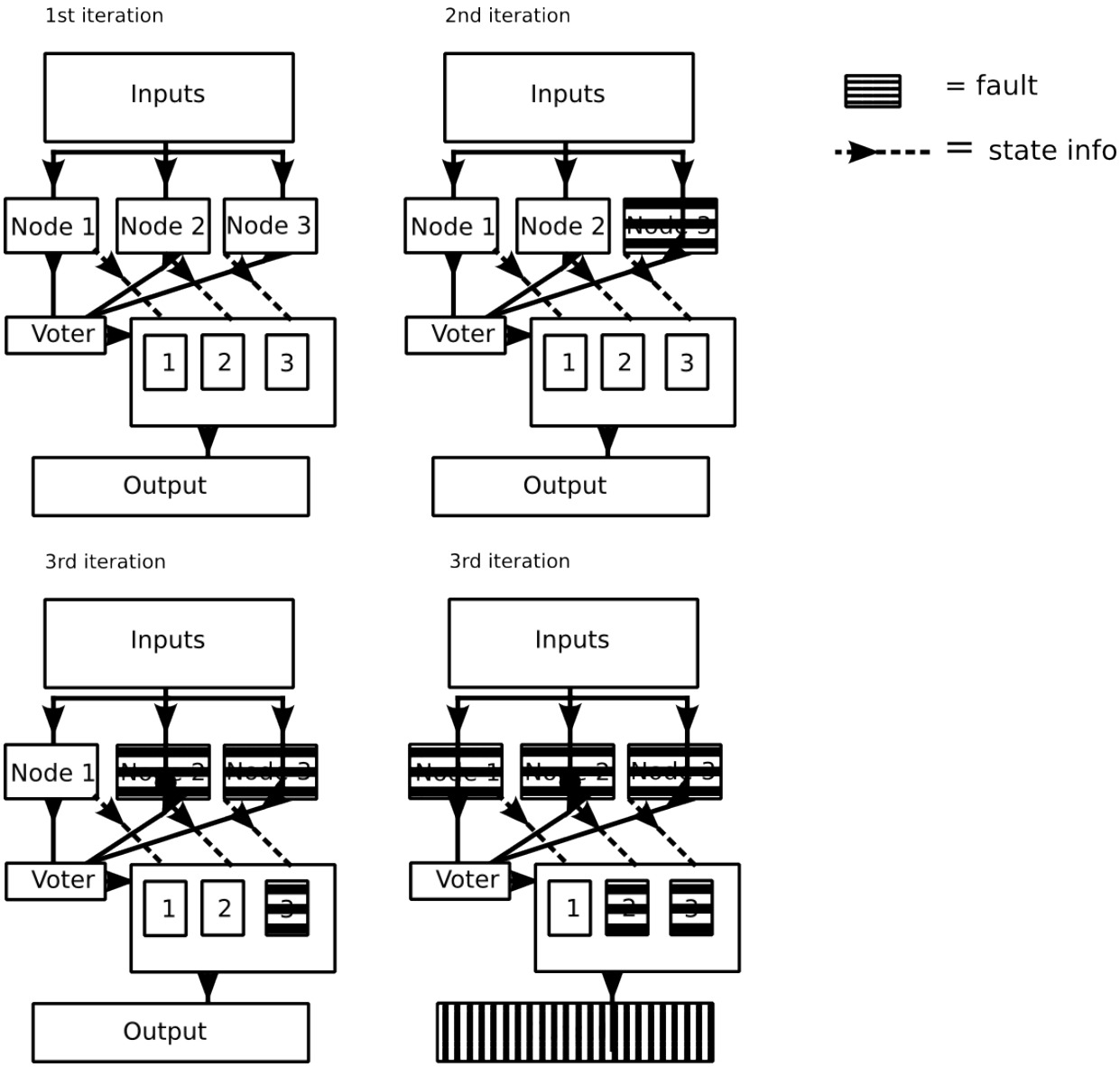


Figure 2.25: Stateful TMR.

this issue, a feature can be introduced that allows for the reliability of a node to be reset, thus avoiding the described situation (Matsumoto et al. [2011]). Depending on the risk of node failure, one of these solutions can be used to improve standard TMR.

2.5.3 DMR checkpointing

This example shows a version of double modular redundancy, coupled with a checkpointing scheme (Yang und Kwak [2010]). There are three important variables being used: the current state, the previous state and the most recent successfully executed state. At regular intervals

(checkpoints), the state of both computing units is stored and compared. If the states match, the current state is stored as the previous state and the most recent successfully executed state, and the system proceeds into the next cycle. If a mismatch is found, where the states of the units are not the same, the action taken depends on the three variables. The process is illustrated in figure 2.26. In the example, unit one suffers from two consecutive transient faults.

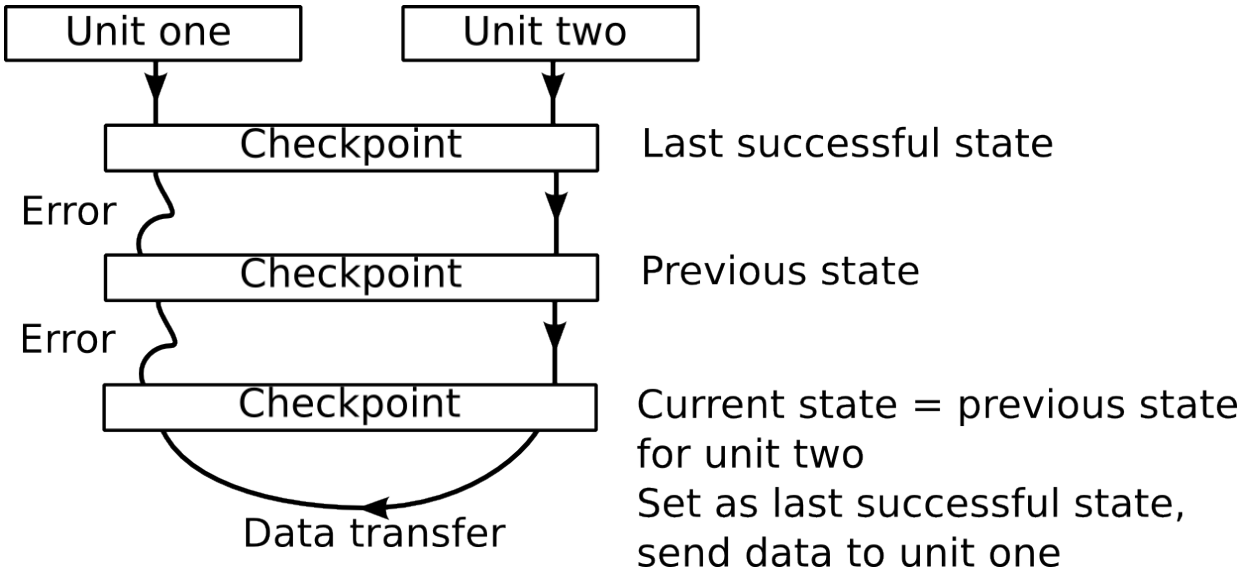


Figure 2.26: DMR checkpointing.

First a comparison is made between the current state and the previous state for each unit, the result of which is sent to the other processing unit in the form of a flag. If a fault has just occurred, the previous and current states are not the same, so both units send flags indicating such a difference. This leads to a rollback to the latest checkpoint, after which both units execute the cycle again. If there are no more faults, execution will then proceed as normal. If the faulty unit experiences another fault, the states are once again compared. Due to the rollback, the previous and current states of the healthy unit are the same. The algorithm then causes the faulty unit to load the state from the healthy unit, after which previous states are updated to the current states and the last successful execution is set as the current one. This is an improvement over a standard rollback scheme, where permanent faults cannot be handled, as a faulty execution would always lead to a rollback (Yang und Kwak [2010]).

2.5.4 Asynchronous checkpointing and rollback

Another example of a checkpointing scheme, here the checkpoints are not global but local - that is, each node has its own set of independent checkpoints. The tasks are assumed to be independent, with no inter-task communication. A dedicated monitor node continously listens for heartbeat messages from the nodes in order to detect a failed node. When such a failure is detected, the monitor uses the latest checkpoint of the node to restart the task on the dedicated monitor node. The advantage of using asynchronous checkpointing is that such a rollback does not cause a rollback for all nodes. In case of a transient fault, the faulty node can, after recovery, check whether its task is running and if so assume the role of the previous monitoring node (Ayav et al. [2008]).

3 Concept

The theoretical concept behind this master thesis gives a detailed view of the ideas behind the implementation of the central task of the project: to provide a redundancy solution for a given, already operational quadrocopter. The task description specifies a set of subtasks which shall be performed within this scope, of which the two most important are the following:

- RODOS utilization
- Hardware design

In order to provide a concept for restructuring the software and modifying the quadrocopter for redundancy, the initial state should be known. This was shown in chapter 2. The nature of the assignment makes it appropriate to divide this chapter into two parts. The first part will describe the restructuring and reconfiguration of the quadrocopter software, focusing on the usage of the RODOS operating system. This will be followed by the hardware concept, which will describe the full hardware redundancy concept to be implemented. An overview of the process of going from the initial state to the final redundancy solution is shown in figure 3.1. In the following sections, the various stages of the process will be described in further detail.

3.1 Restructuring, multi-threading and RODOS

The first step in the restructuring of the software, a step also defined in the task description, is the addition of an underlying operating system, RODOS. This enables support for many functions which will be used later on, and serves as the foundation for the redundant software. Some important advantages that come with the introduction of an operating system (or OS for short) are shown below:

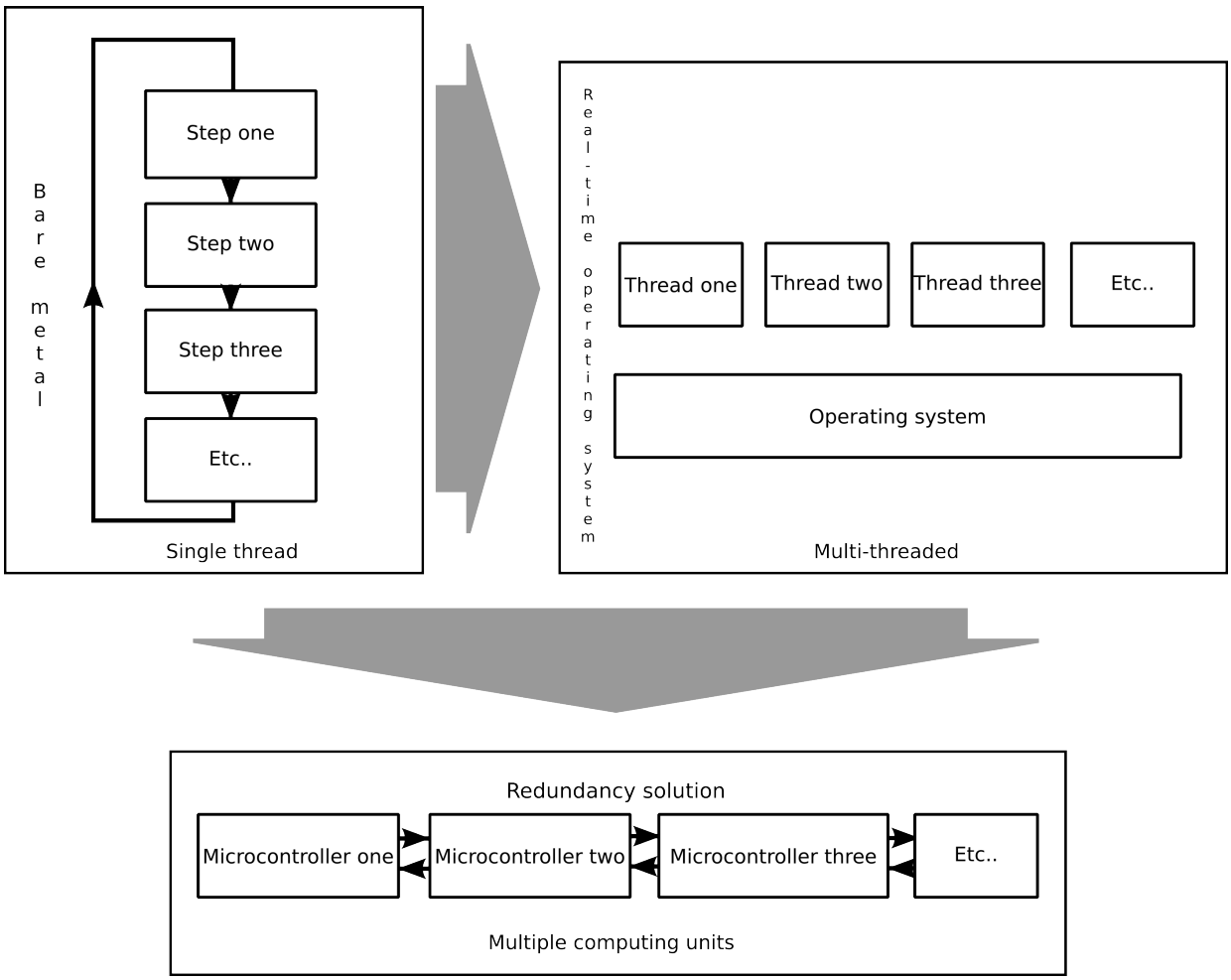


Figure 3.1: Project overview.

- Timing
- Multi-threading
- Modularity
- Portability

In the previous single-threaded system, timing was critical due to a dependence on sample times and actuator timing for the control of the vehicle. With the inclusion of an OS, the importance of timing is further increased. Multiple tasks can execute at different intervals using built-in timing functionality, enabling easier control of task execution compared to a single loop. This is also connected to the next item in the list - multi-threading. As was shown in chapter 2, tasks in separate threads run virtually concurrently, with conflicts and scheduling resolved using

timing functions and task priorities. This greatly simplifies the management of multiple tasks with varying timings.

The possibility of multi-threading also simplifies the management of multiple tasks. Independent tasks can run in different threads, unaffected by non-related tasks in other parts of the software. For tasks requiring data from other threads, synchronization functionality is available, enabling threads to wait for new data and resume once it arrives. Another advantage is that certain tasks can be given higher priority, enabling them to interrupt tasks of lower priority. In the context of this project, this adds much flexibility, as non-flight-critical tasks such as sending debug data can run in the background while flight-critical tasks are given priority. Utilizing multiple threads confers many advantages over a single threaded scheme, and a multi-threaded architecture will be explored during the course of this project.

Multi-threading does not only confer advantages, however. A major drawback, with a severe effect on software development, is the added complexity that multi-threading brings. The current bare-metal, single-threaded design was chosen in part for this particular reason. A program with only a single loop has a deterministic sequence (not including any interrupts), making debugging comparatively easy - one can quickly find the section of code causing problems. With a multi-threaded design, this becomes more difficult. Due to the complex interactions between threads and the simultaneous execution, finding the exact cause of an error is non-trivial, and this is especially true for errors and bugs that occur due to timing of different threads. Any attempt to study the error by standard measures such as the addition of debug print statements will alter the execution time of the thread, thus changing the timing and possibly changing and/or hiding the error that was to be studied. The use of a debugger may have the same effect, changing the execution time and order of the thread. In comparison to a single-threaded system, a multi-threaded architecture will thus require significantly more time for testing and debugging.

3.1.1 Task structure

As was shown already in chapter 2, the quadrocopter software has a set of software modules running sequentially in a loop. These software modules are coupled to varying degrees - sensor readings are coupled to attitude and position determination, which in turn are coupled to the actuator control. The same is true for command input modules, which are used together with the sensor data to give commands to the actuators.

Using RODOS, instead of running all such modules sequentially as part of a main thread, they can be separated into components of a logically minimal size. According to the design philosophy behind the OS (Montenegro [2011]), such tasks could be some of the modules already mentioned: sensor readings, actuator output, communication etc. In order to separate the different modules into independent tasks, the list of modules from chapter 2 is used:

- **Signal processing**
- **Height sensoring & control**
- **Optional modules: Ultrasonic & optical flow**
- **Attitude control**
- **Remote control**
- **Steering control**
- **Send debug values**
- **ADC**
- **Optional modules: Servo control, ultrasonic and infrared obstacle detection**
- **LED toggle**

There is ample opportunity for modularization given the items in the list above. The signal processing task can itself be divided into two separate components - the IO part where the sensors are read, and the signal processing part where calculations are performed. Height sensoring is an independent module, comprised of IO and calculations, and could be separated into its constituent components. The ultrasonic and optical flow modules could also be separated into their own modules.

The attitude control module is similar to the signal processing module, although the situation is the opposite - where the signal processing module takes sensor readings and performs calculations, the attitude control module performs calculations and sends the resultant values to the actuators. Thus, it could also be decomposed into a calculation part and an IO part.

The steering control and remote control modules both handle the input of control values, and could thus be separated into their own components. The sending of debug values is also logically

separated from the other modules; so are the ADC and servo control modules. The ultrasonic and infrared obstacle detection modules could be separated into calculation and IO parts, much like the other similar modules. The last item on the list, the LED toggle, could be separated into its own task; however, since it is only used as an indicator, it would be more sensible to integrate it into the components whose statuses should be shown.

The components of the system described above are shown in figure 3.2. Since the idea behind

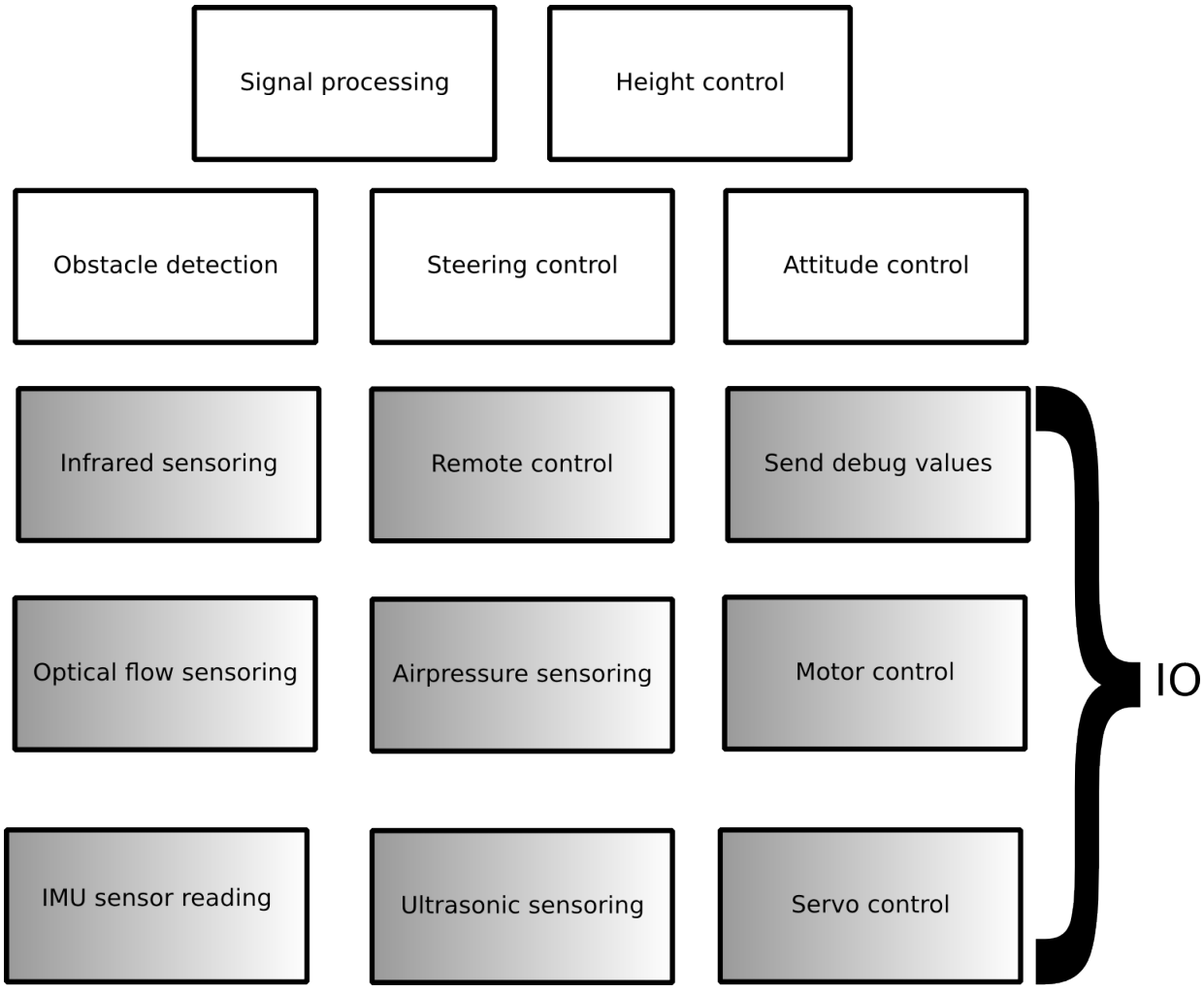


Figure 3.2: Software tasks.

this separation is that the different components should be independent and interchangeable, the old data structure with global variables should not be used. Instead, a new way of storing, sharing and accessing data should be used, which is detailed in the section below. The benefits of separating the software into separate components and tasks are modularity simplicity - all interactions occur via the RODOS middleware, and components can be exchanged at will.

3.1.2 Data structure

The way communication between different tasks and software/hardware components is performed in RODOS is through the use of topics, publishers, subscribers and local storage, as was described in chapter 2. In this section, such a setup will be applied to the components listed in section 3.1.1. In order to do so, it is first necessary to know what communications will occur. In general, all communications in the system can be divided into one of the following three categories:

- Input from sensors / input devices (e.g. UART)
- Internal communications (calculated results, state data etc.)
- Output to actuators / output devices (e.g. UART, LED)

In order to make sensor data freely usable by all components in the system, each input component should publish its data in its own topic, or possibly in a shared topic if there exist multiple sensors of the same type. Topics for internal communication can be published to by one or several publishers; state variables controlling the general state of the craft (engines on, calibration etc) are an example of a topic which would have multiple publishers. Actuator output would normally only have one publisher per type of actuator, in order to not send conflicting commands. For other types of output, such as debugging info or other displays, multiple publishers could be used.

Figure 3.2 shows all the different components in the system. The following topics should be created to let the input devices distribute their data:

- Infrared
- Optical flow
- Inertial measurements
- Ultrasonic
- Remote control
- Airpressure

In order to utilize the topics, they also need subscribers. Of the components listed, the following make use of sensor data:

- Signal processing
- Height control
- Obstacle detection
- Steering control
- Attitude control

Most of these components modify a number of internal variables, such as control data and state data. In order to access this data and also to use it for output, topics should also be created for the following:

- Processed sensor values
- Control data
- State values
- Motor data

The remaining few components provide system output, namely:

- Motor control
- Servo control
- Debug

The motor control gets its values from the attitude control component, and will thus use the motor data topic. For the servo control, processed sensor values are used to give the orientation of the craft in order to keep the platform level. The debug component provides output in the form of debug info, providing the user data about the state of the system and its variables. The debug component should thus subscribe to all important topics in order to receive data for output.

With the information above, an illustration can be made that shows the relationships between different tasks and topics, as well as system input and output. This setup is the software concept for the project, and is shown in figure 3.3. The redesign of the system in this manner enables the creation of a highly adaptable and reconfigurable redundant system. With all communications

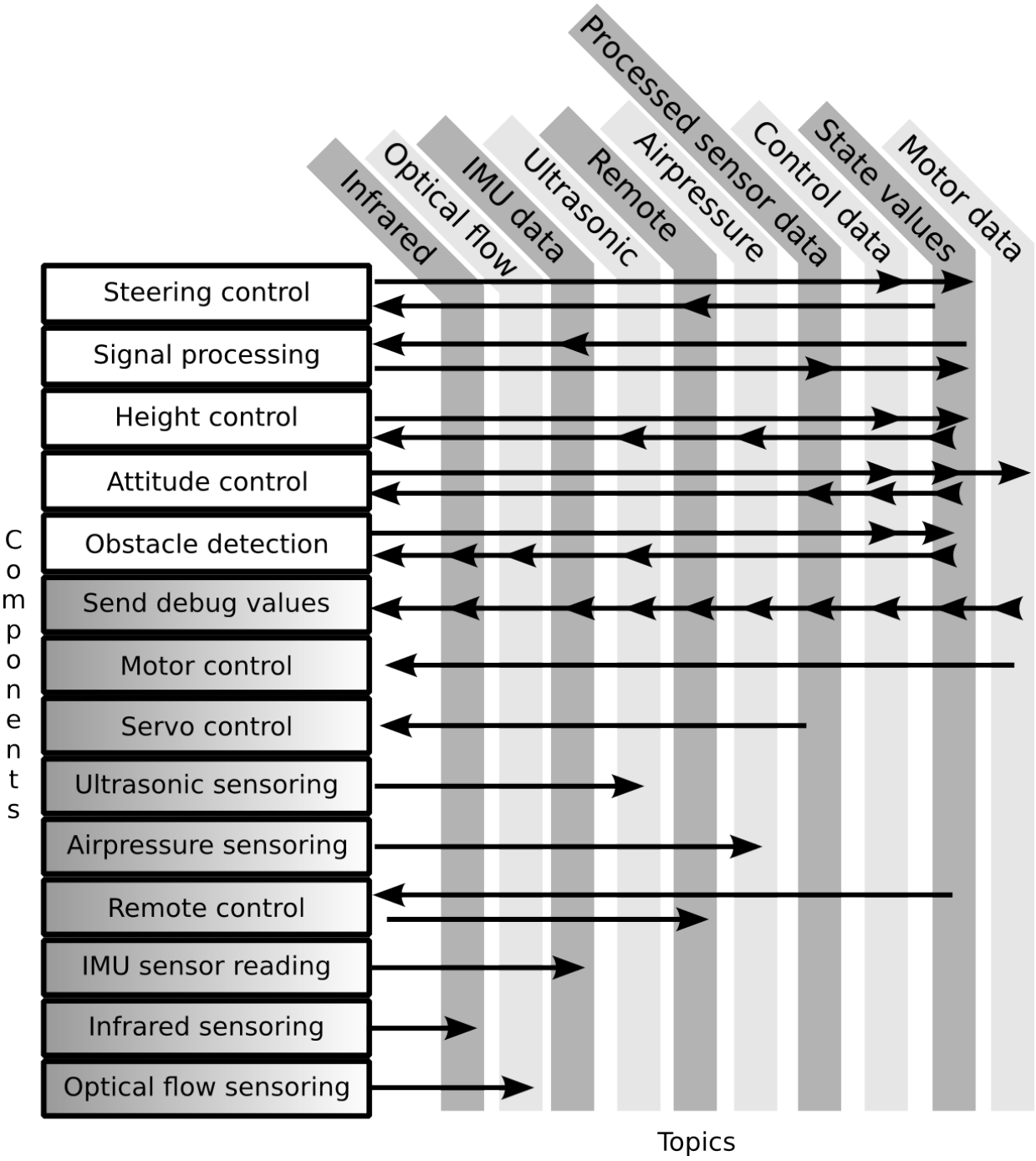


Figure 3.3: Software concept

going through the middleware layer, and all functionality separated into interchangeable tasks, functionality and information can be made to flow seamlessly across node borders using the functionality from RODOS. How this functionality is used is described in the following section.

3.2 Redundancy

Having completed a concept for a redesigned system, the next and final step is to define the concept for the full redundancy solution. As is shown in chapter 2, there are many such solutions available, each with advantages and drawbacks. In order to select the most appropriate solution, selection criteria need to be established. For this concept, the following criteria are used:

- **Simplicity**

  Since the goal of the project is to increase the reliability of the system via the introduction of redundancy, complexity should be as low as possible. An increase in complexity increases the risk of errors, by having a higher chance of unanticipated side effects caused from a lack of overview when designing the system. Increased simplicity, on the contrary, decreases this risk as the interactions of simpler components can be more easily understood and any unwanted side effects minimized. If required, complex functionality can be formed from combinations of simpler, well-understood components.

- **Mass, volume and power**

  Since the quadrocopter has limited physical resources available, weight, volume and power consumption should be kept as low as possible.

- **Adaptability**

  Since the quadrocopter is in a constant state of change, due to being a research project, it is desirable to have a redundancy solution that can adapt to changes in the platform and changes in demands on the redundancy.

Considering the above criteria, an initial decision can be made in order to discard some categories of redundancy solutions. Due to the limited size and power available, any solution which requires four or more separate hardware units can be immediately discarded. Multiple self-checking pairs and four-channel modular redundancy are thus not viable solutions. Remaining are solutions with one, two or three redundant hardware units. The redundancy solution needs to protect against both software and hardware faults, which makes a single hardware unit unviable as well. Separate hardware units are required.

The remaining architectures, involving two or three redundant units, can be selected amongst. In order to do so, it is useful to recollect the different configurations that are available. These are the following, ranging from the simplest to the most complex:

- Dual redundancy with BIT
- Self-checking pair
- Self-checking pair with fault-down
- Triple modular redundancy

Each of these architectures can handle a different type of fault condition. Dual redundancy can handle a single loss of function, whereas a self-checking pair can detect a single malfunction but not handle it. A self-checking pair can both handle a loss of function and detect a malfunction. TMR can protect against both types of faults. At a quick glance, TMR thus seems like a clear winner, as it handles more types of faults than any of the other configurations. However, it also demands three separate units plus a voter, a clear drawback compared to the other redundancy solutions. It also adds complexity to the system, especially in the case of majority voting which requires synchronization. A self-checking pair with fault-down then becomes a good compromise, trading away the full protection of TMR for reduced weight, volume, power requirements and complexity. In addition to this, the use of a watchdog timer with reset capabilities outside of the normal program execution enables the recovery of a failed unit.

To further decrease the likelihood of system failure, another layer of security can be added, in the form of relays. With the above mentioned solution, a faulty unit could still affect the input/output of the system following, for instance, a watchdog reset. In order to prevent this, relays can be added between the processing unit and any input/output, acting as additional BIT. If the unit fails, there is less likelihood of it both keeping the relay active and producing signals that would disrupt sensors and/or actuators. This concept is shown in figure 3.4.

Another version of the above hardware concept is to add additional redundancy in the form of an extra sensor. This prevents a sensor failure from causing a system failure, and thus eliminates the sensor as a SPOF. Such a setup is shown in figure 3.5 and due to increased reliability because of the added sensor, this is the version that will be used in this project.

As for software redundancy, the platform on which the project is built, combined with the required

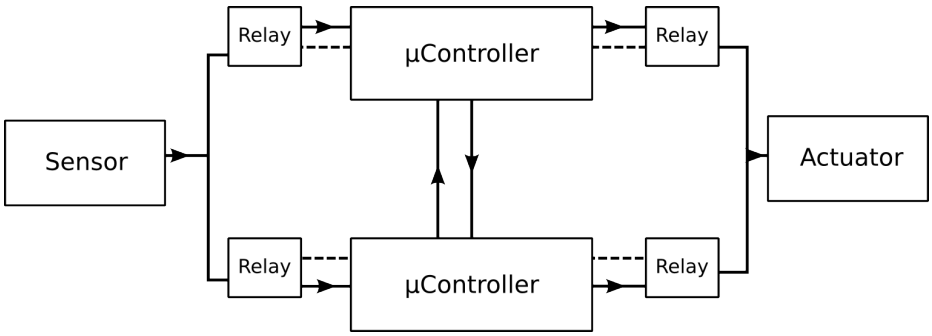


Figure 3.4: Hardware concept.

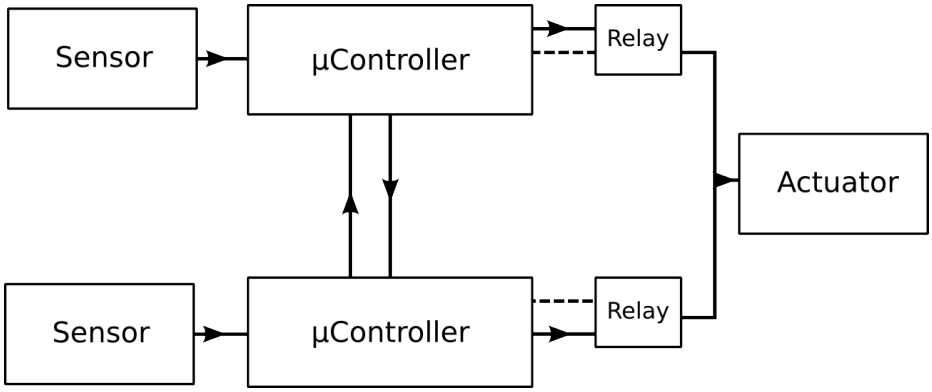


Figure 3.5: Hardware concept with redundant sensors.

sampling frequency, does not allow for multiple calculations of outputs within an acceptable time frame. The restructuring of the quadrocopter software, however, helps achieve redundancy by enabling easier inter-node communications. As was mentioned earlier, topic based communication simplifies modularity - this is also the case for redundancy. Using the middleware layer of the operating system combined with another of its features, gateways, selected topics can be forwarded to other nodes. This allows for easy, reconfigurable synchronization. In the case of a self-checking pair, it also allows for easy error-checking, since all data will be available for comparison.

The final part of the concept concerns adaptability and reconfigurability. The division of the software tasks into threads in section 3.1 also simplifies better resource utilization, by allowing for adaptive load management. Using the topics, nodes can report to each other the number of tasks that are currently running on each node and subsequently distribute the load evenly across all nodes in the combined system.

The modularity that is gained by dividing the software into different inter-communicating threads and tasks allows for greater reconfigurability. This also extends to the redundancy concept

via the use of gateways and inter-node communications. The modular nature of the code allows one to exchange components easily, since all modules use the middleware layer as an interface. For redundancy solutions, reconfiguration requires more effort, but modularity is nonetheless a major benefit. The concept redundancy solution thus also aims to provide the possibility to later redefine the chosen configuration. An example of such a reconfiguration is going from the self-checking pair above to a middle-value selection TMR solution. This would require only a few steps, namely:

- Removing the output function from the calculation nodes and publish the values instead
- Removing the logic controlling which node is in command
- Introduce a voter, with simple functionality that simply compares its input and selects the middle values

4 Implementation

This part of the document will detail the implementation process of both hardware and software, and explain the reasoning behind design choices and any compromises or deviations from the concept that were made. As for the concept, the implementation is divided into two parts - software and hardware. The first part to be described is the software implementation, where the main body of the work has been done. This will be followed by a description of the hardware modifications that were made in order to provide redundancy to the system.

4.1 Software

The software implementation process was performed as a series of iterations, starting with the unmodified quadrocopter software and ending with the final software with support for redundancy. Between these extremes were a number of intermediate steps, which will be detailed below. For each of the iterations, the following will be detailed:

- Software tasks, topics and data structure
- Program flow
- Issues (if applicable)

For the final version, the code that manages the redundancy functionality will be described also.

4.1.1 First iteration

The first iteration of the software was intended as a verifier that the software could run on RODOS, and otherwise did not add any functionality or change the code structure. In order to use the software with RODOS, some small modifications were necessary. The initial code relied heavily

on a timer counter; this was disabled in favor of using the built-in RODOS timer functionality. Also, the software module used to run the on-board display for debugging data was causing the software to become unstable when used with RODOS, and it was therefore disabled in this iteration.

4.1.1.1 Software tasks, topics and data structure

The tasks in this iteration are the following:

- Main thread
- Debug thread
- Remote control thread

The tasks are all implemented as threads of equal priority, using the RODOS Thread class.In addition to the tasks listed, this iteration also uses an object from the RODOS Initiator class, whose code will be run before any threads are initialized or start running. This configuration is illustrated in figure 4.1
Since the code was not modified besides being adapted to run on RODOS, there are no topics used and the data structure is the same as earlier, using global variables for accessing shared data.

4.1.1.2 Program flow

The software first starts with RODOS initialization. Early in this process, the init() methods of all Initiator objects are called. Thus, for this version, the first user-implemented code to run is the code in the Initiator object. There, the hardware initialization that is described in section 2.1 is performed, setting up all hardware drivers and interrupts, as well as quadrocopter - groundstation communications.

Following this, further on in the RODOS startup process the init() methods of all threads are called. The Debug and Remote threads have empty initialization methods, so the only code that is run in this step is the initialization code in the Main thread. This code is the software initialization that was described in chapter 2.1; Kalman and mean filters are initialized, as well as attitude and height control. Sensor and remote calibration are also performed in this step.

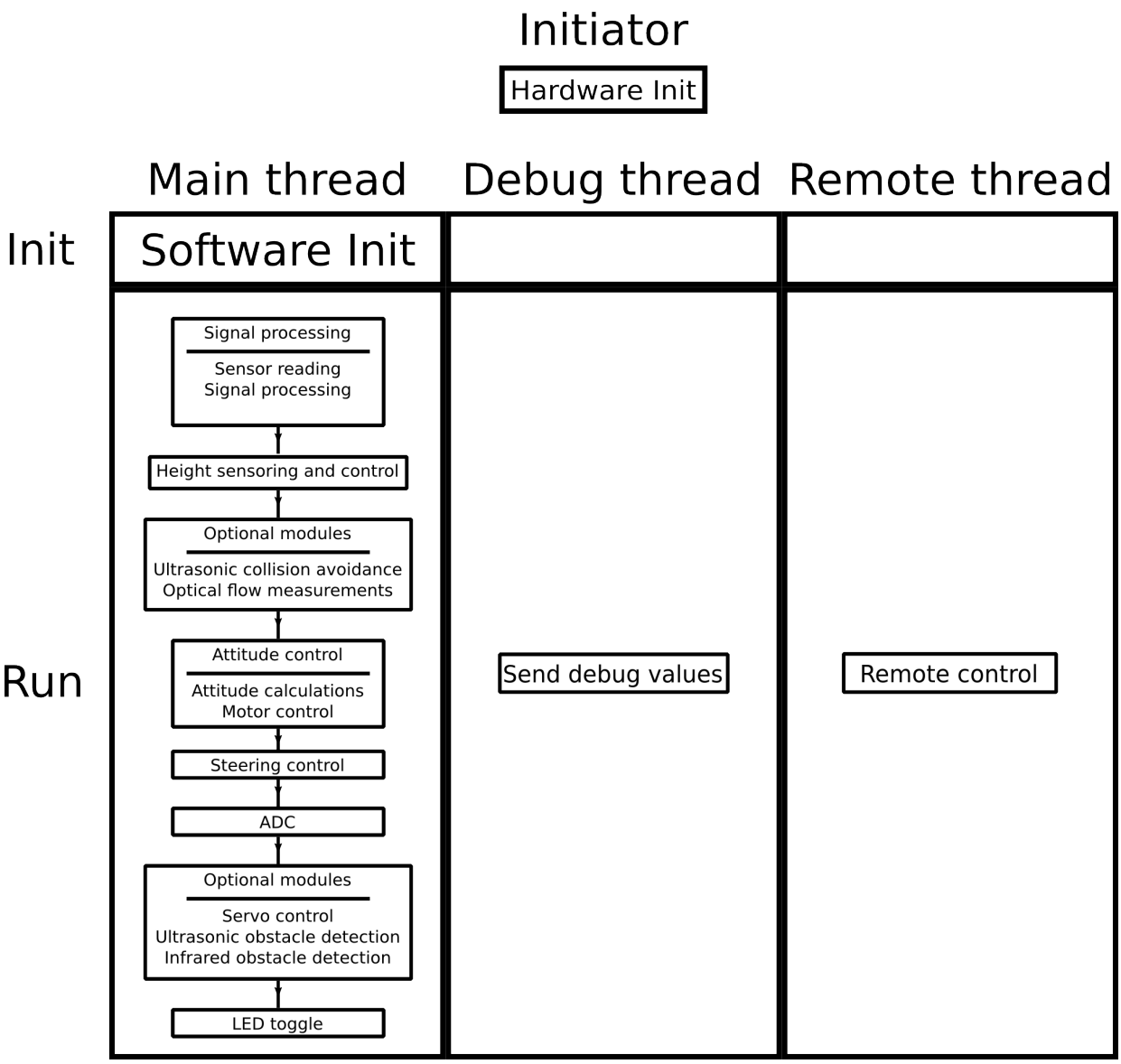


Figure 4.1: First software iteration overview.

In the next step, the scheduler starts scheduling the different tasks, allotting CPU time according to priority. Since all tasks have the same priority in this iteration, time will be equally shared. The Remote and Debug threads have simple run() methods, with only one method each. The Remote thread reads and parses data from the remote UART, and the Debug thread sends debug data over the debug UART. The Main thread, then, implements the remaining functionality of the Main loop as described in section 2.1, in the same sequential order.

4.1.1.3 Issues

When running the code on the quadrocopter, there was an issue with jittering. The controls and motor actuation were jittery when compared to the non-RODOS version of the software, leading to worse handling and performance. One possible cause could be the combination of global variables and multiple threads accessing the same data, and issue that was described in chapter 2. However, as this iteration had only the purpose of verifying that the code would run on RODOS, the issue was not investigated further. Instead, the implementation proceeded to the next step, where the functionality of the RODOS middleware is implemented.

4.1.2 Second iteration

The second iteration directly builds upon the first iteration, with two major differences. Instead of using global variables for shared data, the different tasks employ local storage and the publisher/subscriber protocol for data transfer. Enabled by this transition, the different software modules of the previous main loop are now also divided into separate, independent tasks as was described in chapter 3. Furthermore, in order to focus on developing the framework necessary for the later implementation of redundancy, the optional modules are not implemented. Their inclusion is not critical for the implementation of the restructured software, nor for the redundancy, and they can be reimplemented at a later time when the functionality of the redesigned system is verified. The disabled tasks are shown in the list below.

- Height sensoring & control
- Ultrasonic & optical flow sensoring
- ADC
- Servo control, ultrasonic and infrared obstacle detection

4.1.2.1 Software tasks, topics and data structure

The tasks that are implemented in this iteration are shown in the list below. The functionality of each task is virtually the same as for the similarly named software modules that are described in chapter 2 and shown in figure 2.2. There are two differences: the signal processing module is split

into a sensor reading task and a signal processing task, and the attitude control module is split into an attitude calculation task and a motor control task which sends commands to the motors.

- IMU sensor reading thread
- Signal processing thread
- Attitude control thread
- Motor control thread
- Debug thread
- Remote thread
- Steer control thread
- LED thread (former Main thread)

The initiator from the previous iteration was left unchanged, and has the same functionality in this version. The tasks are implemented with varying priorities, shown in the table below. A higher number indicates a higher priority.

- 110: LED, Debug
- 120: IMU Sensor reading
- 130: Remote, steer control
- 140: Signal processing
- 150: Attitude control
- 200: Motor control

This configuration is illustrated in figure 4.2
This iteration introduces a full utilization of the RODOS middleware. The list below shows the different topics that are implemented and their publishers and subscribers.

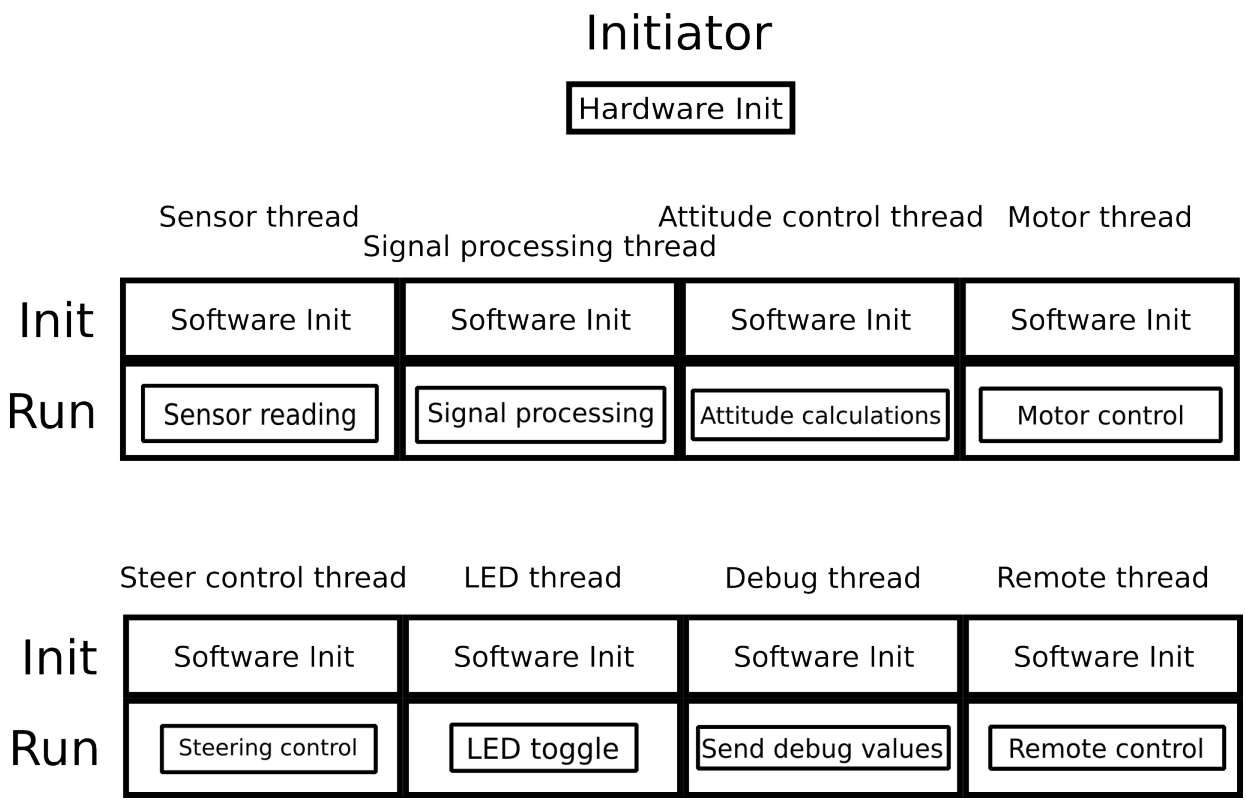


Figure 4.2: Second software iteration overview.

- **Processed IMU sensor data**

  Publishers: Signal processing thread

  Subscribers: Control thread, debug thread

- **IMU sensor bias data**

  Publishers: Sensor thread, signal processing thread

  Subscribers: Debug thread, signal processing thread

- **Raw IMU sensor data**

  Publishers: Sensor thread

  Subscribers: Signal processing thread

- **Control data**

  Publishers: Control thread, steer control thread

  Subscribers: Control thread, debug thread, motor thread, signal processing thread, steer control thread

- **Quaternion data**

  Publishers: Signal processing thread

  Subscribers: Debug thread, Control thread

- **Master state values**

  Publishers: Sensor thread, steer control thread

  Subscribers: Debug thread, motor thread, sensor thread, steer control thread

- **Remote data**

  Publishers: Remote thread

  Subscribers: Debug thread, steer control thread

- **Kalman data**

  Publishers: Kalman filter

  Subscribers: Signal processing thread

4.1.2.2 Program flow

Like the previous iteration, the software starts with the RODOS initialization and hardware initialization via the initiator. The next step, however, is different. There is no longer a main thread to running the software initialization and calibration; rather, each thread manages the initialization of its own variables. This initialization is performed according to the previously shown priorities. With such a setup, modularity increases as the initialization procedure is only run if the task is included. Some data is also shared at this stage; the sensor read thread publishes the sensor bias values gained from the sensor calibration, which are stored by the signal processing thread.

Following initialization, the thread scheduling begins, as in the previous iteration. Here, there are two types of threads: fixed period and event-driven. The fixed period threads are the debug thread, remote thread, LED thread, steer control thread and IMU sensor reading thread. The debug and remote threads have a similar implementation as earlier, executing the same method at a fixed interval. The difference in this implementation is that they receive data through the RODOS middleware, instead of through global variables. All other functionality is unchanged. The LED thread is a simple indicator that the system is running, toggling an LED at a regular

interval. This shows that there is some spare processing capacity left since it has the lowest priority.

The steer control thread receives control data and remote data and sets the mode of control of the quadrocopter. This is done at a regular interval, less frequent than the sensor readings as the mode of control will not change frequently.

The motor control thread has the highest priority and executes at a fixed interval. Being a time-critical component, actuator output takes precedence over sensor input and any other processing. The reasoning behind this setup is that it is safer to possibly execute the motor control with values from the previous cycle, than to prioritize new values with the risk of a delay in motor actuation commands which would cause, for example, a loss of altitude. The data used for motor commands will, in the worst case, not be older than one cycle before new data is received from the attitude control process.

The final thread that executes at a regular interval is the IMU sensor read, which reads sensor data from the IMU and publishes it. The act of publishing raw sensor data acts as a trigger for the remaining event-driven threads.

The signal processing thread listens for raw sensor data through the middleware; every time there is data published in the raw data topic, the signal processing thread is activated. After using this data to calculate processed sensor values and attitude information, the processed data is published. This in turn activates the attitude control thread, which listens for processed data. It uses this to produce motor commands, which are published to the corresponding topic for use by the motor control task.

4.1.2.3 Issues

The major issue with the second iteration was instability - the software would freeze at seemingly random intervals. After debugging, the problem could be narrowed down to a likely cause, namely thread scheduling and variable access. Although the method of sharing data between different modules in the software changed with the introduction of the RODOS middleware, the actual containers for the data did not change. Sensor data, control data, state data and so on are stored in large structs, with many different types of data per struct. The sensor data struct, as an example, contains both raw data, bias data and processed sensor data. The situation is analogous for the other types of structs. When these structs are passed around using the topics, all data is sent,

regardless of which fields were actually updated. This could lead to unwanted behaviour in combination with thread scheduling. Take the following example:

- Control data is read by the steer control thread
- The steer control thread updates part of the struct
- The scheduler performs a thread switch
- The control thread reads control data, updates a part of the struct, and publishes it
- The steer control thread resumes, and publishes the control struct

In the last step, the steer control thread publishes old control data together with its own modifications. If another thread later uses control data, it will not see the updates from the control thread. Thus, changes were necessary to split the structs into smaller components that can be updated in one task, and a third iteration was required.

4.1.3 Third iteration

The third iteration of the software changes mostly how data is passed to the different tasks. Another change is a consolidation of functionality; in the second iteration, the execution of sensor reading, signal processing, attitude control and motor control would always be sequential, even though they were located in different tasks. Since they are logically connected, and to ease the task of debugging, in this version they are consolidated into one central task which handles all high-frequency operations. Another advantage of this setup is an execution speed gain; data no longer needs to be passed around in the same quantities as before. A disadvantage is that modularity is decreased; changing any of the central modules now requires more work than simply switching a component, and they also cannot be changed at run-time.

4.1.3.1 Software tasks, topics and data structure

The following tasks are implemented in this iteration:

- Central thread
- Debug thread

- Remote thread
- Steer control thread

As was explained above, the core functionality is consolidated into the central thread, which now contains all functionality for sensor reading, signal processing, attitude control, motor control and the LED activity indicator. The other threads maintain their functionality from the earlier versions. Another change is the removal of the initiator; its functionality is implemented in the RODOS user main function, which executes after thread initialization but before any thread executes. This is done as to have better access to the various threads, in preparation for the adddition of redundancy functionality. The initialization procedures are instead moved to the first, non-looping part of the central thread. The reason for this is also related to redundancy preparations; the execution of such initialization needs to be controlled by another thread which implements the redundancy, in order to have different initialization functions for primary and secondary nodes. This configuration is illustrated in figure 4.3

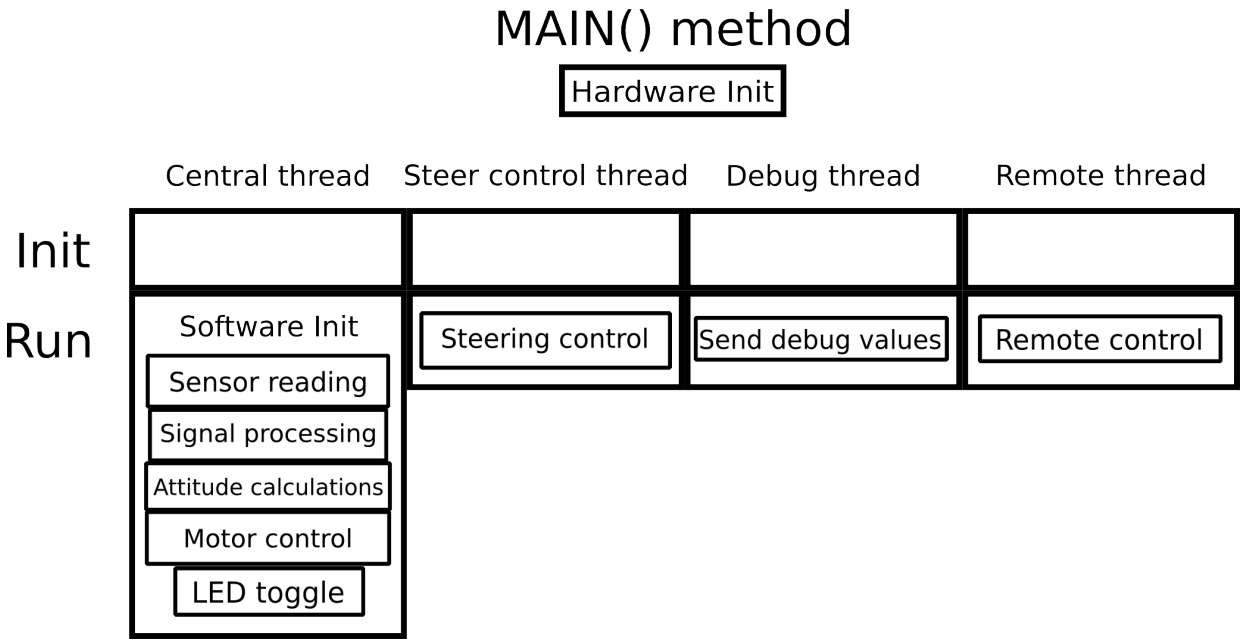


Figure 4.3: Third software iteration overview.

Although the number of tasks is smaller in this iteration compared to the previous version, the number of topics is greater. This is done as a solution to the issues with the second iteration. The topics, publishers and subscribers implemented in this version are the following:

- **Gyro & acc sensor data**

  Processed gyro and accelerometer data from the IMU

  Publishers: Central thread

  Subscribers: Central thread, debug thread

- **Sensor bias data**

  Sensor bias data for the IMU

  Publishers: Central thread

  Subscribers: Debug thread

- **Kalman data**

  Kalman filter variables

  Publishers: Central thread

  Subscribers: Debug thread

- **RPY sensor data**

  Processed roll, pitch and yaw values

  Publishers: Central thread

  Subscribers: Debug thread

- **Height sensor data**

  Height data from the height sensors

  Publishers: Central thread

  Subscribers: Debug thread, steer control thread

- **Raw sensor data**

  Raw sensor output from the IMU

  Publishers: Central thread

  Subscribers: N/A (redundancy preparations)

- **Quaternion data**

  Quaternion data output from the signal processing module

  Publishers: Central thread

  Subscribers: Debug threads

- **RPY control data**

  Roll, pitch and yaw value output from the attitude control module

  Publishers: Central thread

  Subscribers: Debug thread, steer control thread

- **Desired RPY control data**

  The desired roll, pitch and yaw values set either by remote control or by the autonomous control

  Publishers: Central thread

  Subscribers: Debug thread

- **Height control data**

  Height value output from the height control module

  Publishers: Central thread

  Subscribers: Debug thread, steer control thread

- **Desired height control data**

  The desired height value set either by remote control or by the autonomous control

  Publishers: Central thread

  Subscribers: Debug thread

- **Throttle control data**

  Throttle control value from either the remote control or the autonomous control

  Publishers: Central thread

  Subscribers: Debug thread

- **Motor control data**

  Motor control values set by the attitude control module for each motor

  Publishers: Central thread

  Subscribers: Debug thread

- **Position control data**

  Position control data for autonomous navigation

  Publishers: N/A (for later reimplementation)

  Subscribers: Steer control thread

- **Flight control data**

  FLight control data for autonomous navigation

  Publishers: N/A (for later reimplementation)

  Subscribers: Steer control thread

- **Desired RP flight control data**

  Desired roll and pitch data set by the autonomous flight control module

  Publishers: N/A (for later reimplementation)

  Subscribers: Steer control thread

- **Desired height flight control data**

  Desired height data set by the autonomous flight control module

  Publishers: N/A (for later reimplementation)

  Subscribers: Steer control thread

Besides these topics, the master state variable struct has been divided into its components, in order to let any task or driver set the value of such a state variable without interfering with other state variables.

As can be seen, the increase in topics is considerable. This is due to the need to separate the structs into smaller, logically connected entities, where each is only published by a single task if possible. The separation also serves the purpose of preparing the system for later reimplementation of disabled functionality.

4.1.3.2 Program flow

The program flow in the third iteration is a combination of the previous two iterations. The central thread approximates the main thread of the first iteration, with all shared data passed via local thread variables. The remote, debug and steer control threads share data via the RODOS middleware, as in the second iteration.

The central thread has the highest priority and the highest frequency. At a set interval, sensor reading, signal processing and attitude control are performed sequentially. Debug data is sent independently at a lower frequency and priority; remote data is gathered at half the sample frequency of the central thread, and has a higher priority than the debug thread. The steer control executes every five sample periods, and has the second highest priority.

4.1.3.3 Issues

The third iteration exhibits good characteristics, except for one important characteristic: execution speed. An investigation into the matter showed that the publishing of data and consequent reads by the steer control thread using the middleware layer was a probable cause. A new version was thus required in order to correct this issue.

4.1.4 Final iteration

The final iteration further consolidates the code, and merges the steer control thread with the central thread. It also introduces the redundancy task, which is be detailed below.

4.1.4.1 Software tasks, topics and data structure

The following tasks are implemented in the final version:

- Central thread
- Debug thread
- Remote thread
- Redundancy management thread

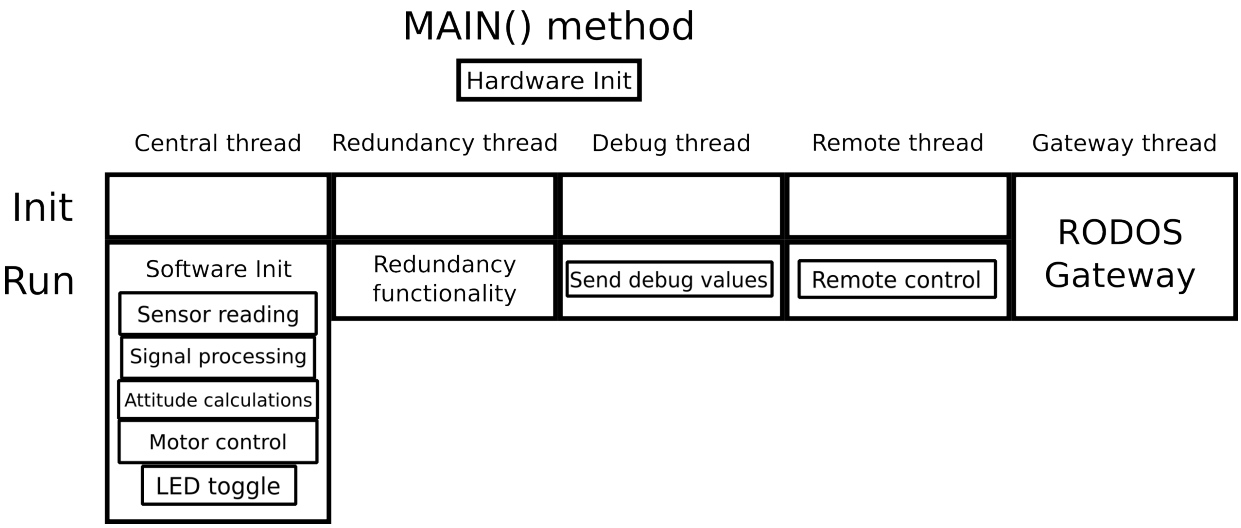


Figure 4.4: Final software iteration overview.

This configuration is illustrated in figure 4.4

The topics in this version of the software are largely the same as in the third iteration. There are no new topics; however, some topics are disabled since they are not used. The following topics are disabled:

- **Position control data**
- **Flight control data**
- **Desired RP flight control data**
- **Desired height flight control data**

4.1.4.2 Program flow

The program works almost identically like the previous version regarding the first three tasks. The main difference is that the steer control module now uses local thread variables instead of the publisher/subscriber protocol to access its data. However, a most significant addition exists in the form of the redundancy management module, which is explained in its own section below.

4.1.4.3 Redundancy management module

The redundancy management module handles the fault tolerance of the system. It is comprised of the following parts:

- **An internode communications module (RODOS gateway)**
- **Primary/secondary detection functionality**
- **Recovery functionality**
- **Watchdog functionality**

At system startup, the module starts before any other user task. As a first step, the primary/secondary detection functionality is activated. The node listens for so-called *heart beat* signals, in order to determine its status. If such signals are found, the node designates itself as a secondary node and waits for an initialization package. This consists of quaternion orientation data, in order for the secondary unit to start with correct orientation information. When such a package is received, the node then proceeds to periodically accept orientation data from the primary, while also listening for heart beat signals. If a heart beat signal is missed, a flag is set to indicate the missed heart beat. If the flag is active and a heart beat is missed, the node assumes that the primary node has failed and assumes command - the procedure for doing so is explained below. Any received heart beat will clear the flag. In contrast, if there are no heart beat signals heard at startup, the node assumes the role of primary and listens for initialization requests, as well as periodically sending heart beat signals.

All internode communications are performed using the RODOS Gateway class and UART link interface. This enables any topic to be selected for forwarding to other nodes in a network via the specified UART interface - in this implementation, UART 1 of the microcontroller is used at a baud rate of 56700, and there is only one other node connected. As part of the startup process, quaternion data and heartbeat signals are set to be forwarded.

The watchdog functionality is using the watchdog timer to make sure that a node is reset if it freezes, thereby also turning the power off for its relays and protecting the system from faulty behaviour. At startup, the watchdog timer is initiated and set to a timeout period of 285 ms. As long as the node receives raw data from its sensors, the watchdog timer is reset every time the redundancy management task runs. This serves a dual purpose - if the sensor stops functioning, the node and sensor are reset, clearing any transient faults. If the entire node freezes, the watchdog timer ensures that it is reset and does not cause problems for the healthy node. To this end, a separate initiator object is included, which disables the watchdog timer early in the startup process. This enables a shorter timeout period.

The last part of the functionality is the recovery process, when two heartbeats are missed in a row. The secondary node then assumes the primary role and activates its output relays. It also starts publishing heart beat signals and listening for initialization requests.

4.1.4.4 Issues

The initial watchdog timeout period was set to approximately 500 ms, which was too high to ensure continuous operation in the case of failure, and the timeout period was thus lowered. The watchdog functionality of the microcontroller only allows certain discrete timeout periods, and the next available timeout period was 285 ms. This was chosen as the timeout period of the system, with the side effect that a crashed node would no longer be reactivated. System initialization requires more than 285 ms, and the failed node thus remains in a reset state until the system has a hard reset or is manually reset. This trade-off was required in order to have a fast enough failover time, with the consequence of turning a transient fault into a permanent fault. However, the failed node cannot disrupt the operation of the healthy node while in a reset state, and the fault is thus contained.

4.2 Hardware

The hardware implementation features two different versions. For the first three software iterations, the hardware is the same as for the initial configuration. The setup is illustrated in figure 4.5.

For the redundancy solution, the implementation follows the concept. Of the two architectures shown in chapter 3, the version with two sensors is used. This is done to add extra dependability in case of complete sensor failure. The redundant setup is shown in figure 4.6.

The following sections explain the different added components of the redundant hardware.

4.2.1 Communication

The communication between nodes is performed using UART 1 of the microcontroller at a baud rate of 56700. Data from the remote is shared via a common connection to UART 2 - the data is only read, and no protection is required. The output from the remote control receiver is therefore split, and then connected to the UART 2 receive pins on both microcontrollers without any relays.

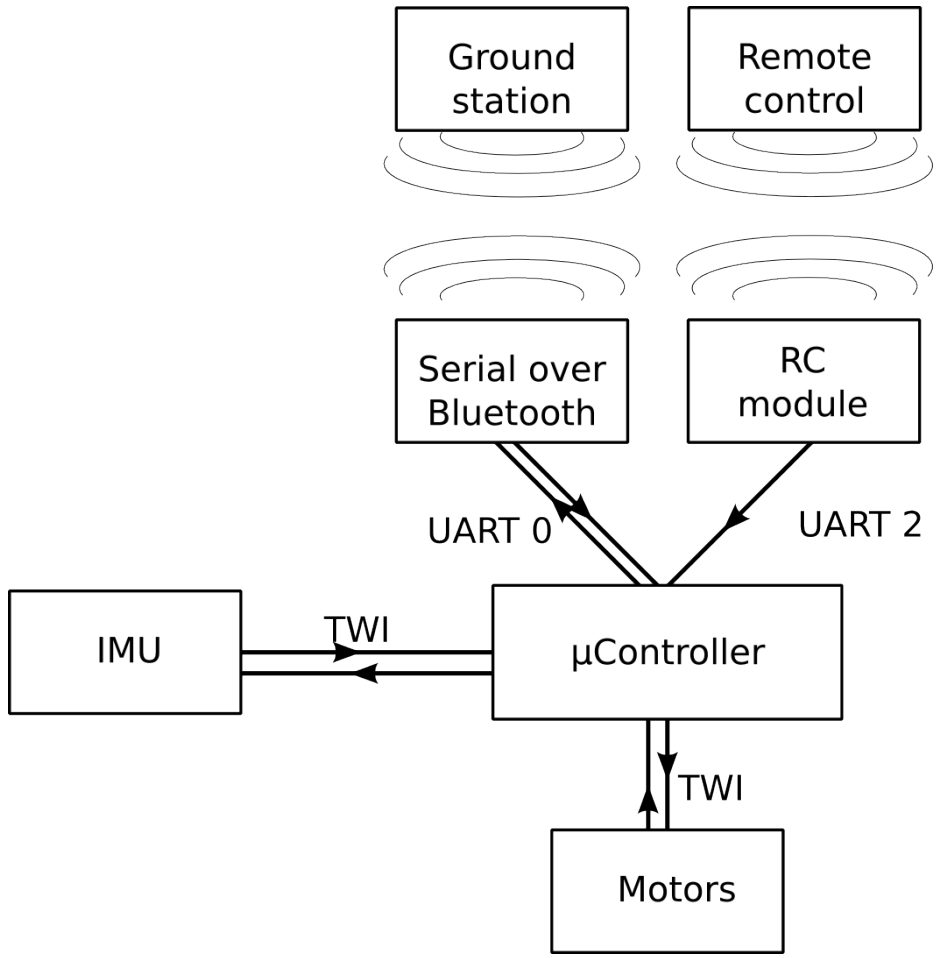


Figure 4.5: Non-redundant hardware.

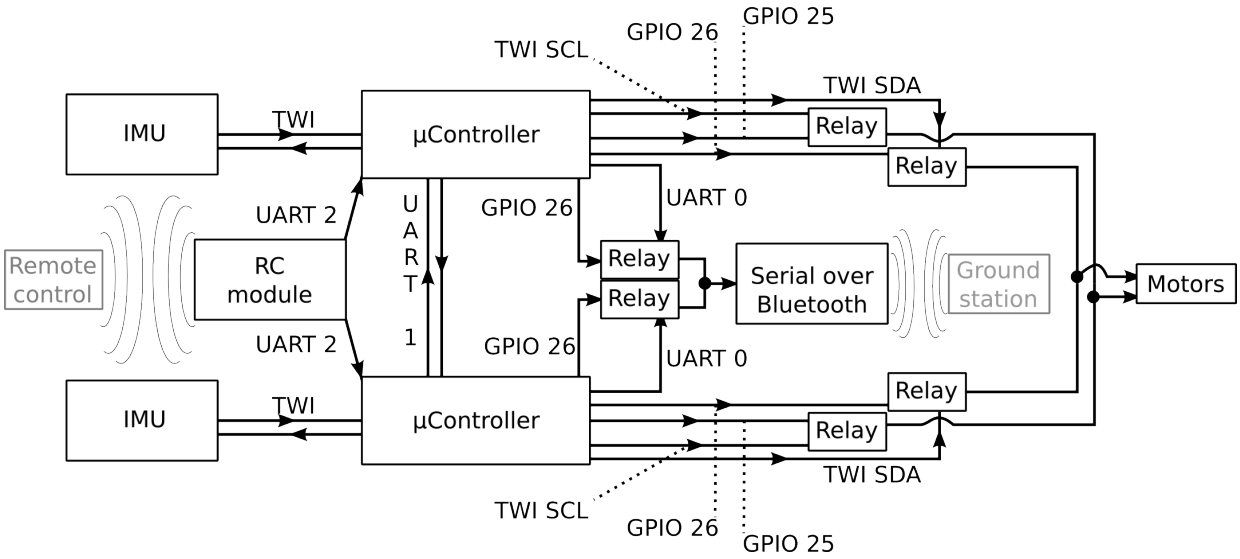


Figure 4.6: Redundant hardware.

The debug output is sent using UART 0 over Bluetooth for both microcontrollers. Therefore, this connection requires protection. If both microcontrollers had simultaneous access to the debug

output, the transmitted signal would be corrupted and the debug data would be invalid. Thus, the UART 0 transmit pin of each microcontroller is routed through two separate relays (one for each microcontroller). The relays are activated in two ways: either at startup, when a node cannot hear any heartbeat signals and thus assumes the role of primary, or when a secondary node fails to detect two heartbeat signals in a row and assumes command following a failure of the primary node. Thus, only one node at a time can send debug output, ensuring that a faulty node does not produce noise. The activation of the relay requires only the use of a single GPIO pin in order to provide power; in this implementation, GPIO pin 26 is used.

4.2.2 Redundant sensors

As illustrated, two identical IMU units are used, each connected to its microcontroller via TWI. Of the two concept solutions shown in chapter 3, the version with two sensors was selected in order to provide additional redundancy. For the version with only a single sensor, a sensor failure would result in a catastrophic system failure, as neither the primary nor the secondary node would have any access to sensor data. This would then cause the quadrocopter to lose control and subsequently crash.

Using the solution with two sensors, however, this scenario is avoided. If either a node or its sensor fails, the redundant node and its sensor can take over and keep the quadrocopter operational. This setup also allows for an additional layer of redundancy in future implementations. Each sensor can be connected to both microcontrollers, thereby allowing for a simultaneous failure of the primary node and the redundant sensor, and vice versa.

Since there is one sensor per microcontroller, there is no need for sensor access to be protected by relays. Each microcontroller has a direct TWI connection to its sensor and simultaneously receive sensor data.

4.2.3 BIT via relays

The relays in the illustration are part of the BIT of the system, ensuring that only a system that is healthy enough to specifically set the GPIO pins necessary for driving the relays can transmit data. There are four relays protecting the connection to the motors, two per microcontroller. The TWI SDA and SCL lines of each microcontroller are connected to its sensors, and also to the

inout side of the relays. The output lines of the relays of each microcontroller are joined, SDA with SDA and SCL with SCL, and the combined lines are connected to the motors. In order for a microcontroller to pass output to the motors, its relays therefore have to be activated. The conditions and method of activation are the same as for the relay protecting the debug output, and the GPIO pins used for relay activation are pins 25 and 26.

5 Evaluation

The evaluation of the implemented solution was, like the implementation, performed in a number of steps corresponding to the different system iterations. The early tests during the first three iterations were performed to check the basic functionality of the system and verify that it was producing output matching the initial setup. For the earliest software tests, a static test stand was used, which is shown in figure 5.1.

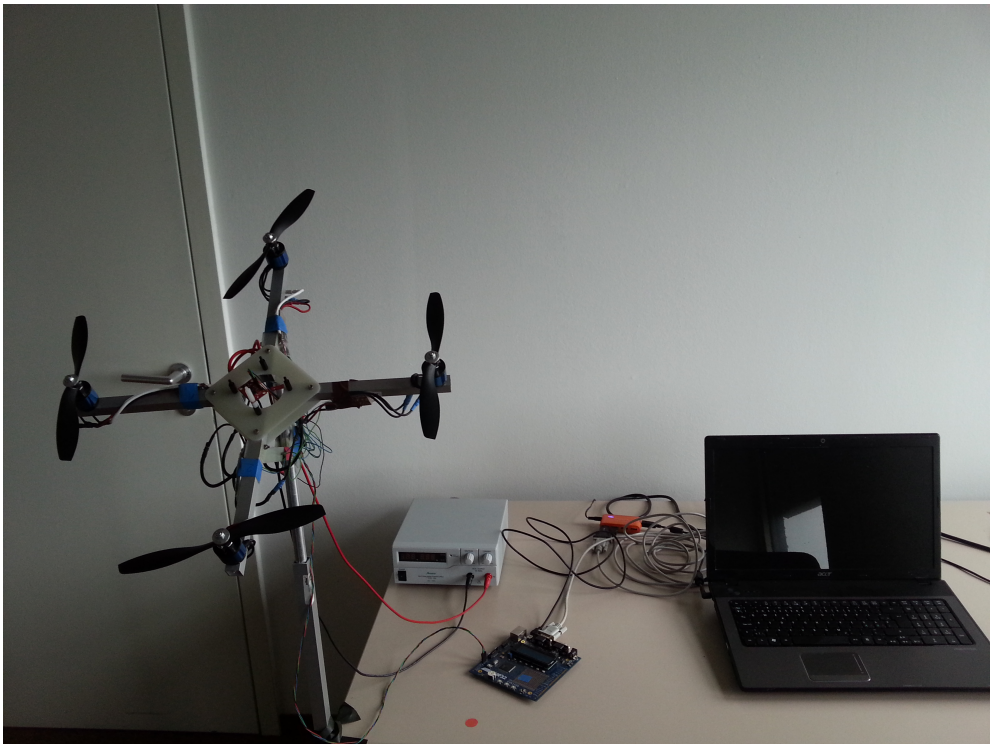


Figure 5.1: Test stand for static software tests.

The test stand was used for initial pre-flight tests in order to verify that the system was receiving correct sensor input and producing correct actuator output. Sensor readings and actuator output values, except for visual inspection of the attitude of the test rig and rotors, were collected via UART and subsequently parsed and displayed using ground station software developed at the

Chair of Informatics VIII at the University of Würzburg. The parts of the interface of this software that were used for testing are shown in figures 5.2 and 5.3.

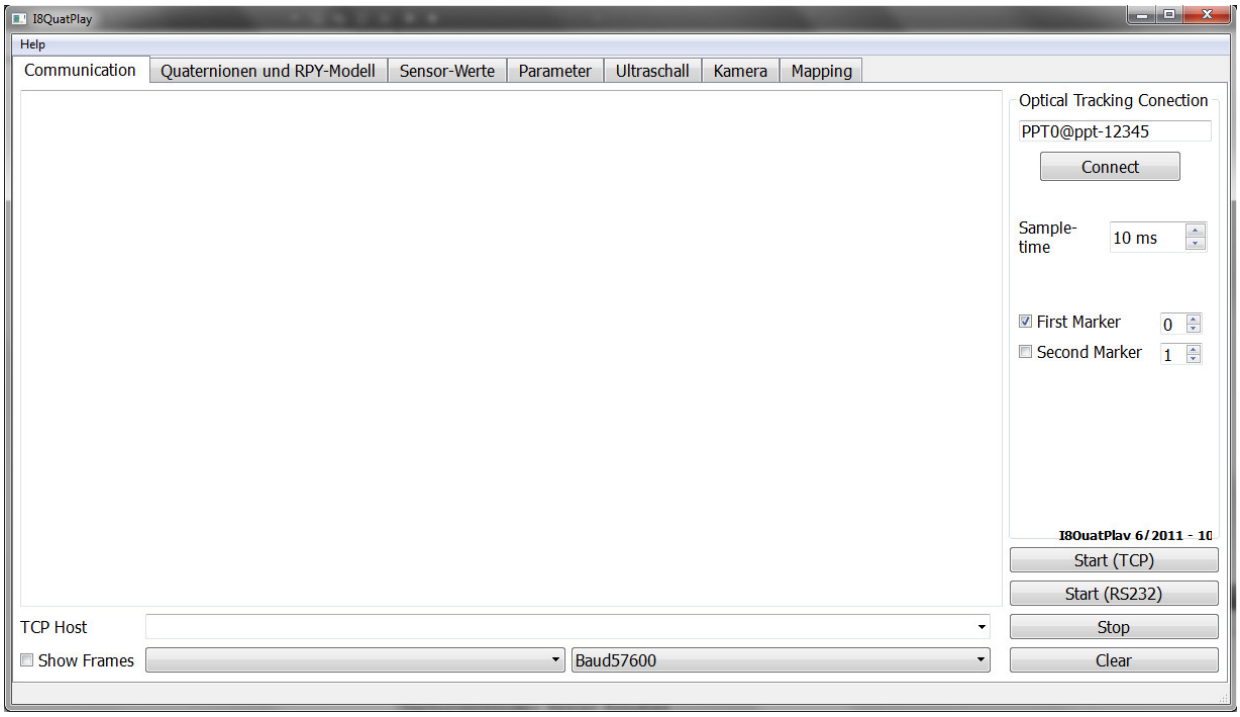


Figure 5.2: The ground station communication interface.

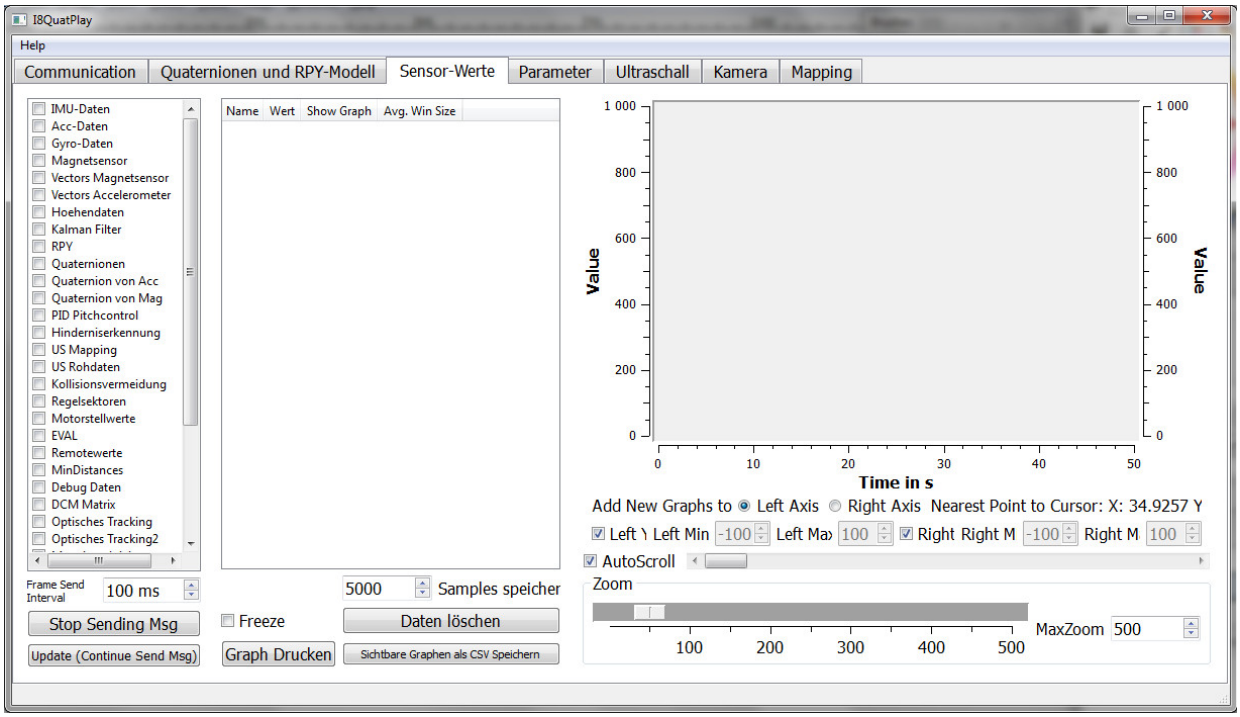


Figure 5.3: The ground station sensor data interface.

For the first iteration, the static tests showed expected system behaviour similar to the initial

software configuration, and the software was then tested using the full hardware configuration, shown in figure 5.4.



Figure 5.4: The standard, non-redundant hardware configuration.

With the full configuration, data for the ground station was gathered via UART over Bluetooth. Although the system could fly, maneuvering quality and responsiveness were not equal to the initial configuration. Significant jitter was present, but as described in chapter 4 the iteration was only meant to test that the code could run on RODOS and the jitter issue was not investigated further in the first iteration.

The second iteration of the software was only tested using the static test stand. Although the system produced normal output, it was prone to seemingly random crashes. Therefore, after initial tests and subsequent debugging, the instability of the system was deemed too great to perform any tests using the actual flight hardware. The data gained from the static tests was instead used in the design of the third iteration of the software.

The third iteration was tested using both the static test stand and the full hardware configuration. Both types of testing showed correct outputs and inputs; however, as described in chapter 4, the speed of the execution was too low which lead to the fourth and final iteration of the software. For all the early tests, the hardware configuration did not change from the initial setup.

5.1 Final iteration, non-redundant hardware

After verifying a sufficient execution speed using the static test stand, the final software iteration was tested using the non-redundant hardware. Early during the testing, the remote control was found to cause disturbances to the system - when the remote was switched on, the software would freeze after a seemingly random period of time. Although sensor data was still transmitted to the ground station, the motors remained switched off and could not be restarted. Debugging data showed that the raw values read from the remote control receiver were rapidly fluctuating, emulating rapid activation/deactivation of the motors. This in turn led to the malfunction of the motor control. A closer inspection of the relevant system parameters gave the source of the fluctuations and thus also the malfunction: a mismatch in clock frequencies between the original quadrocopter code and the system settings in RODOS. After matching the clock frequencies, the system performed as expected. The jitter that was seen in the first tests was no longer present, and maneuvering was equivalent to the unmodified system.

The graphs in figures 5.5 and 5.6 show sample sensor output from the flight test that was performed. The data shown is raw data from the gyro sensor, showing continuous operation for over five minutes. After verifying the functionality of the software on the standard hardware, the next tests were then performed on the redundant hardware.

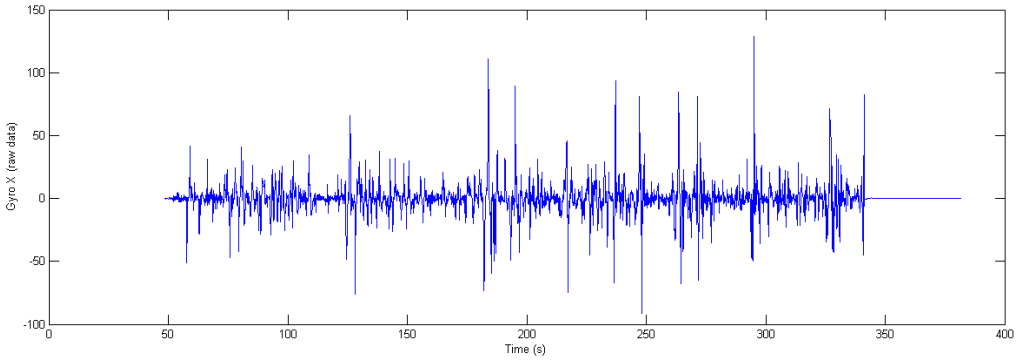


Figure 5.5: Gyro sensor X-axis output, showing continuous operation.

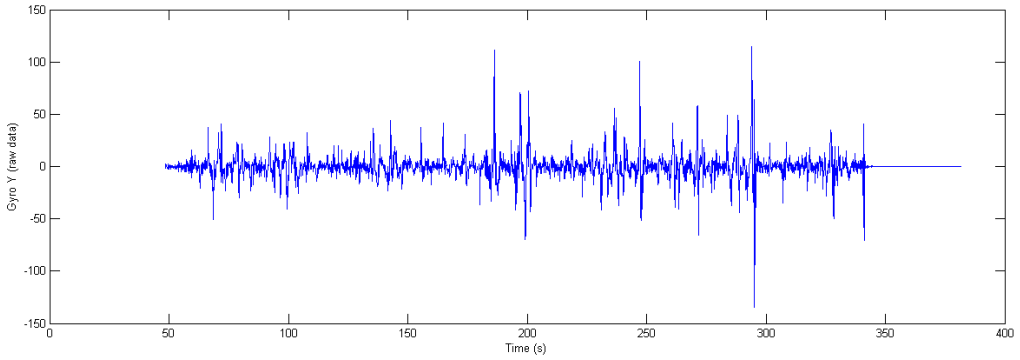


Figure 5.6: Gyro sensor Y-axis output, showing continuous operation.

5.2 Final iteration, redundant hardware

In the final tests, the full redundant hardware configuration was used. This configuration is shown in figure 5.7.

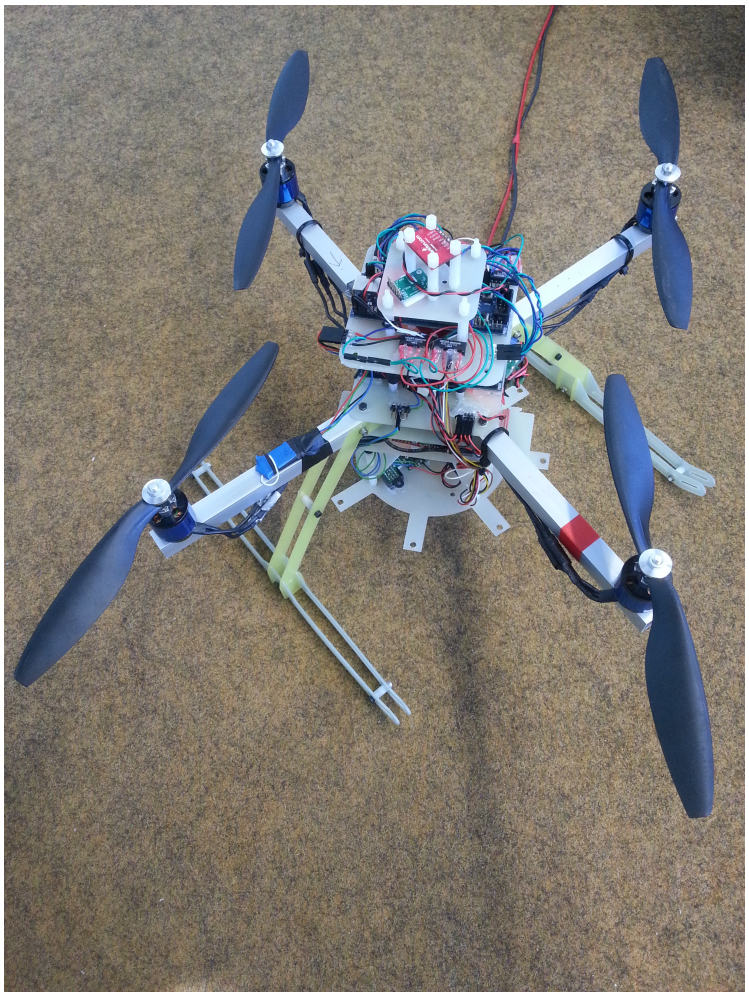


Figure 5.7: The redundant hardware configuration.

In order to test the functionality of the failsafe solution, a simple software task was inserted into the system with the purpose of rendering the primary microcontroller inoperable. The task was set to activate at 30 seconds after system startup, causing the system to freeze. This was achieved by setting the task to the highest priority of all tasks in the system, including system tasks, and letting it run in an infinite loop. In order to simulate both a complete failure and a freeze followed by a watchdog reset, two versions were used. One version of the task deactivated all relays and reset the watchdog at every iteration of the loop, simulating a complete failure. The other version of the task instead looped until the watchdog reset was activated, simulating a software freeze. For the remainder of this chapter, the first version of the task will be referred to as task A and the second version as task B.

For all tests, the TWI communication was a major cause of problems. Following a primary node failure, the secondary would detect the failure in every test that was performed. However, the continued operation of the quadrocopter was often disturbed by problems with the TWI bus. Following a failure, the devices on the bus would often time out due to being busy or not responding, causing a reset of the secondary node and thus a system failure. When the motors are active, the problem becomes more frequent. The likely cause of these problems is that a transmission is interrupted by the failure of the primary node, thereby locking the bus. This problem and possible solutions are discussed further in chapter 6. The results below are taken from the tests where there were no disruptions caused by the TWI bus.

The first tests using the redundant hardware were performed using task A with motors switched off, followed by tests performed using task B with motors switched off. The last tests were performed with motors switched on. Tests using task A and task B showed virtually identical behaviour, with the motors having the largest effect on the results.

The two graphs in figures 5.8 and 5.9 show typical resulting quaternion and RPY data from the tests with motors switched off.

The quaternion data in figure 5.8 shows no evidence of the transition to the secondary node. The RPY data in figure 5.9, however, clearly shows a difference in sensor output after the failure of the primary node. The deviation in the roll axis value is the greatest, with smaller deviations for the pitch and yaw axes. There is also a clear difference in sensor jitter, with the roll axis showing the most.

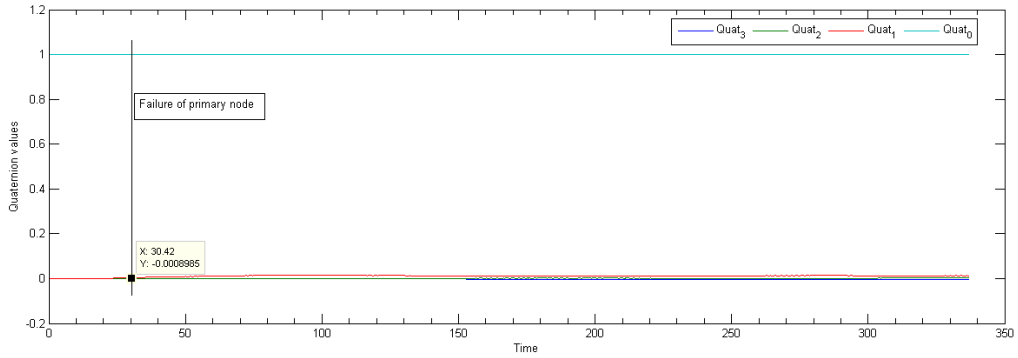


Figure 5.8: Quaternion test data using task A with motors off.

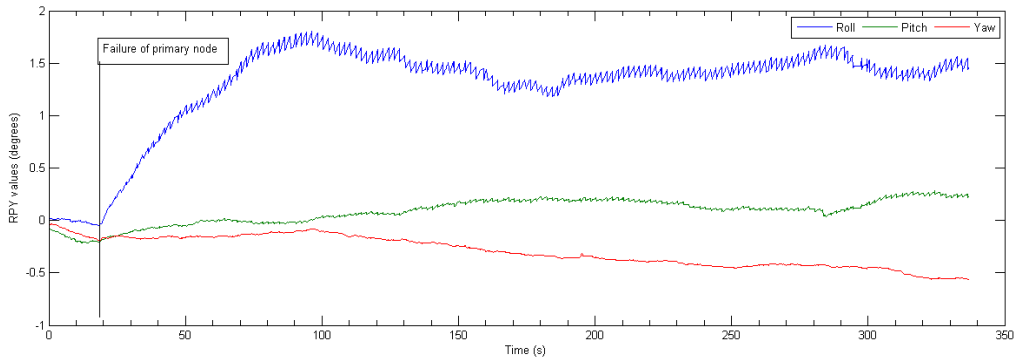


Figure 5.9: RPY test data using task A with motors off.

There were also deviations from the typical behaviour, where sensor data in a small minority of cases would deviate significantly more when switching to the secondary node. This type of deviation is shown in figures 5.10 and 5.11.

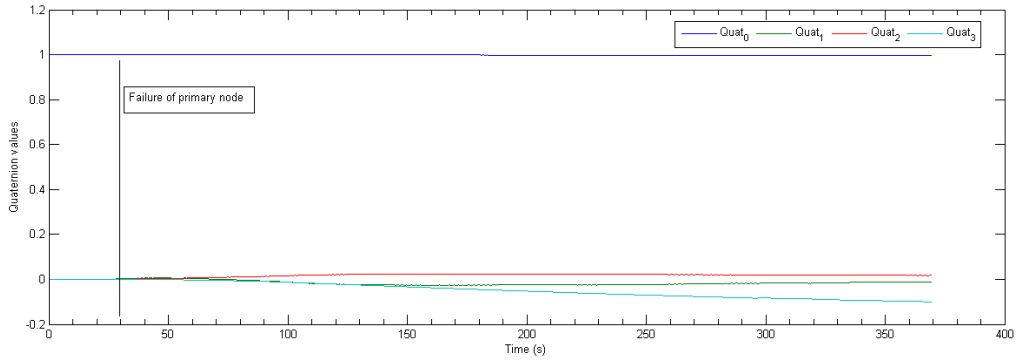


Figure 5.10: Quaternion test data using task B with motors off.

The RPY data in particular shows a large deviation from the nominal value. Possible causes of this deviation and solutions to the problem are discussed in chapter 6.

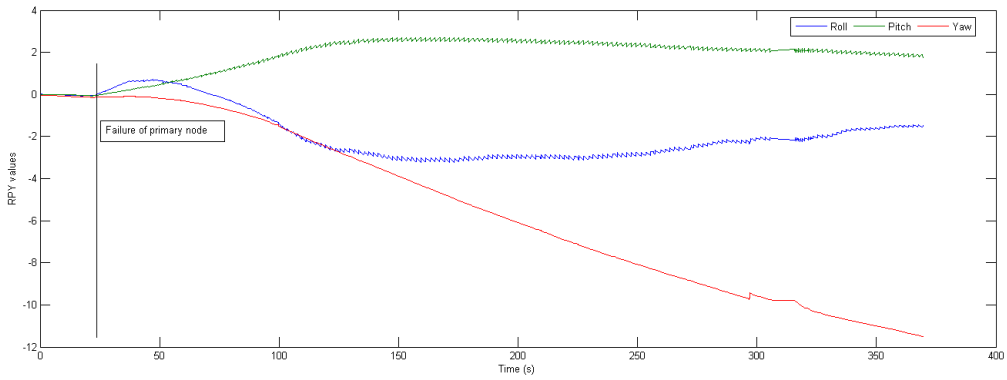


Figure 5.11: RPY test data using task B with motors off.

The final tests were performed using task B with motors switched on, to evaluate the performance under the conditions of a normal flight. The results from these tests are shown in figures 5.12 and 5.13.

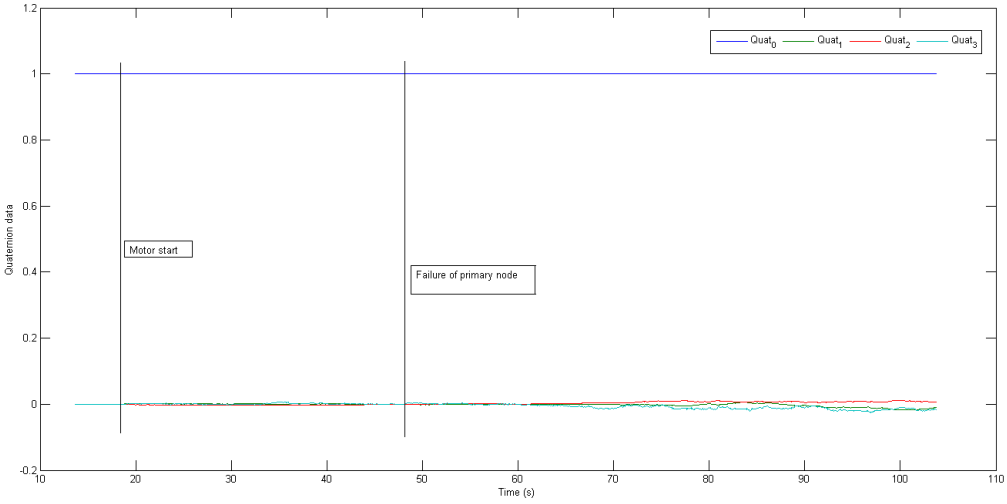


Figure 5.12: Quaternion test data using task B with motors on.

The quaternion data shows no change following the primary node failure. The RPY data, however, shows a highly significant change in behaviour. The likely cause of this change and a solution to address the visible jitter are discussed in chapter 6.

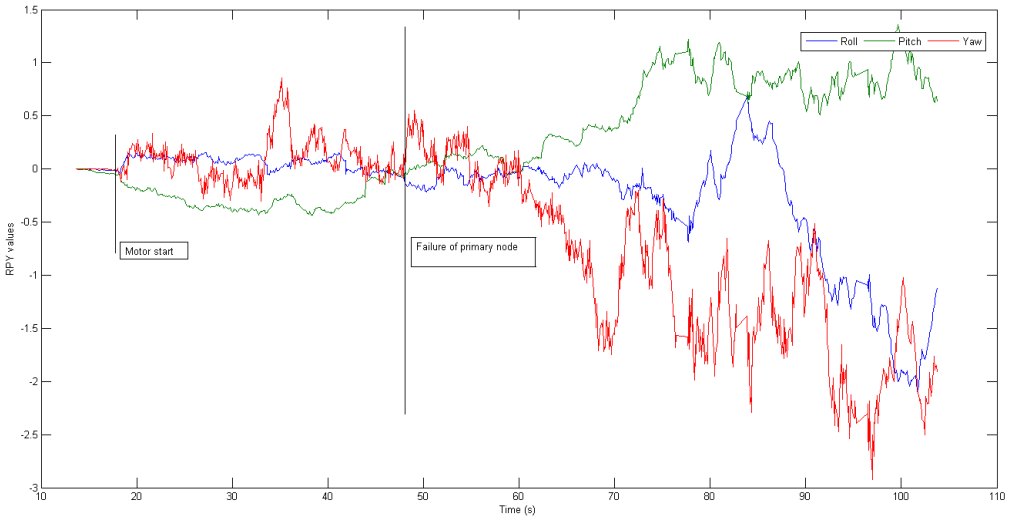


Figure 5.13: RPY test data using task B with motors on.

5.3 Additional testing

In addition to the tests above, a short additional test was performed in order to investigate the slower speed of the software which was seemingly caused by the shift to using the RODOS middleware and local storage, instead of the previous global variables. An extra task was added with a lower priority than any other task, in order to measure how much free processing time was available. This task simply incremented a counter for a set period of time and displayed the result. By comparing the numbers displayed by the counter for two different versions of the software, one using global variables and one using the middleware layer and local storage, a rough estimate of the overhead could be made. Repeated tests showed that on average, the software version using global variables was 15 to 17 percent faster.

6 Discussion and outlook

The discussion and outlook part of the thesis will be divided into two sections, starting with a discussion of the obtained results and any problems that occurred, together with an outlook and suggestions of future work. This is followed by a summary of the project and the final conclusions made.

The first aspect of the results to be discussed is the influence of the TWI bus, since it is a major cause of errors. As was mentioned in the previous chapter, the secondary node would detect a failure, but then reset due to problems with the TWI. These errors are not solved by stopping, re-initializing or resetting the bus, rendering such software solutions ineffective.

A likely cause of the problem is that a transmission on the bus is interrupted when the primary node fails, causing one of the TWI slave units to lock the bus until the transmission is finished. When the secondary node then tries to use the bus, the sensors and/or motors do not respond. The observed behaviour supports this idea, as the TWI problems occur more frequently when the motors are active and there is more traffic on the bus, and thus more transmissions that can be interrupted.

A future solution to this problem could be a hardware implementation where the microcontroller can perform a hard reset of the TWI slaves, thus resetting the bus and enabling continued operations. Another possible way of solving the problem would involve using a different means of communicating with the motors for the secondary node, bypassing the problem with using TWI altogether. Such a solution would even be favorable from the perspective of redundancy, as the dissimilar modes of communication would provide less risk of a common mode failure. However, it would also require more extensive hardware modifications.

As for the data from the tests, it shows some irregularities when the primary node fails. When the motors are switched off, the node switch can easily be seen in the graphs. Despite this, in the majority of tests the data from the axis which deviates the most shows a maximum deviation

of less than two degrees, which is acceptable for flight. The rarer cases where data for one axis deviates by more than ten degrees before stabilizing are more severe, and need to be further investigated. A possible cause of this behaviour, and also of the more common lesser deviations, is that all state data is not transferred when the secondary node is updated. An idea for future investigations is to increase the amount of data that is transferred, especially including Kalman filter variables and other state variables.

In the case when the motors are switched on, the behaviour becomes even more irregular, with larger variations and noise for all axes of measurement. There is an easy explanation for this phenomenon, however; the physical layout of the redundant hardware. As the non-redundant hardware of the quadrocopter was already fixed at the start of the project, the room for redesign was limited. As a result, all the redundant hardware, including the microcontroller, sensor and relays were placed on top of the original hardware. While such a setup is statically stable, the activation of the motors causes problems. The material used for the structure on which the components are added is flexible; when the motors start, they induce strong vibrations in the structure upon which the intertial sensor rests. The resulting output from the sensor if therefore noisy. One solution to this problem could be to add additional structural support, thus making the entire structure more rigid.

Another way to prevent this problem, when a final redundancy solution is decided upon, is to move the redundant sensor lower and closer to the axis of rotation of the quadrocopter. Besides reducing the noise in sensor output due to vibrations, such a move also serves another purpose. In the current redundant configuration, any rotation also causes the accelerometer of the redundant IMU to register an acceleration. This problem could be avoided if the sensor were placed closer to the centre of the quadrocopter.

Another idea for future work is the reimplementation of all previous functionality, adapting all the software modules of the standard configuration to work with RODOS. Following a reimplementation, the possibility of task distribution across the two nodes could be fully taken advantage of. Due to the modular nature of the system, other configurations could also easily be experimented with. The implementation of a TMR solution would require only small changes in the software, and error checking functionality could easily be added in future implementations by the use of the RODOS middleware layer.

Overall, the configuration presented in this thesis provides a modular, adaptive redundancy solution for quadrocopters. The software developed during the course of this project demonstrates the viability of using RODOS in the context of quadrocopter redundancy, and how it enables easier communications and integration between multiple nodes. However, there are also some issues that need further investigation. Speed is one such issue, as seen in chapters 4 and 5. The 15 percent speed difference shown in tests can not fully explain the observed slowdown in the third iteration, however, and further investigation is necessary. Related to the speed issue is also the inter-node communication, as it is based on the middleware layer. In order to be able to transfer higher amounts of data between nodes, the speed issue must first be resolved.

There is still work to be done with TWI communications in order to provide a fully stable solution; however, once that work is done, the solution presented here can either provide fault tolerance as is or easily be expanded for future implementations.

7 Bibliography

[Ahlstrom et al. 2002] AHLSTROM, K. ; TORIN, J. ; FERSAN, K. ; NOBRANT, P.: Redundancy management in distributed flight control systems: experience simulations. In: *Digital Avionics Systems Conference, 2002. Proceedings. The 21st* Bd. 2, 2002, p. 13C3–1–13C3–7 vol.2

[Ayav et al. 2008] AYAV, Tolga ; FRADET, Pascal ; GIRAULT, Alain: Implementing fault-tolerance in real-time programs by automatic program transformations. In: *ACM Trans. Embed. Comput. Syst.* 7 (2008), August, Nr. 4, p. 45:1–45:43. – URL `http://doi.acm.org/10.1145/1376804.1376813`. – ISSN 1539-9087

[Aysan et al. 2008] AYSAN, H. ; PUNNEKKAT, S. ; DOBRIN, R.: VTV - A Voting Strategy for Real-Time Systems. In: *Dependable Computing, 2008. PRDC '08. 14th IEEE Pacific Rim International Symposium on*, 2008, p. 56–63

[Bolchini et al. 2002] BOLCHINI, C. ; POMANTE, L. ; SALICE, F. ; SCIUTO, D.: A system level approach in designing dual-duplex fault tolerant embedded systems. In: *On-Line Testing Workshop, 2002. Proceedings of the Eighth IEEE International*, 2002, p. 32–36

[Bolduc 2001] BOLDUC, L.P.: X-33 redundancy management system. In: *Aerospace and Electronic Systems Magazine, IEEE* 16 (2001), Nr. 5, p. 23–28. – ISSN 0885-8985

[Bolduc 2002] BOLDUC, L.P.: Clock synchronization in an N-modular redundant system. In: *Digital Avionics Systems Conference, 2002. Proceedings. The 21st* Bd. 2, 2002, p. 9A5–1–9A5–8 vol.2

[Chandhrasekaran und Choi 2010] CHANDHRASEKARAN, V.K. ; CHOI, Eunmi: Fault tolerance system for UAV using hardware in the Loop Simulation. In: *New Trends in Information Science and Service Science (NISS), 2010 4th International Conference on*, 2010, p. 293–300

[DLR 2013] DLR: *RODOS: Real time kernel design for dependability*. 09 2013. – URL http://www.dlr.de/irs/en/desktopdefault.aspx/tabid-5976/9736_read-19576/

[Gostelow 2011] GOSTELOW, K.P.: The design of a fault-tolerant, real-time, multi-core computer system. In: *Aerospace Conference, 2011 IEEE*, 2011, p. 1–8. – ISSN 1095-323X

[Hamamatsu et al. 2010] HAMAMATSU, M. ; TSUCHIYA, T. ; KIKUNO, T.: On the Reliability of Cascaded TMR Systems. In: *Dependable Computing (PRDC), 2010 IEEE 16th Pacific Rim International Symposium on*, 2010, p. 184–190

[Hammett 2001] HAMMETT, R.: Design by extrapolation: an evaluation of fault-tolerant avionics. In: *Digital Avionics Systems, 2001. DASC. 20th Conference* Bd. 1, 2001, p. 1C5/1–1C5/12 vol.1

[Hammett 2002] HAMMETT, R.: Flight-critical distributed systems - design considerations. In: *Digital Avionics Systems Conference, 2002. Proceedings. The 21st* Bd. 2, 2002, p. 13B3–1–13B3–8 vol.2

[Janarthanan und Gherbi 2011] JANARTHANAN, V. ; GHERBI, A.: Architectures for Dependable Software Systems. In: *Information Technology: New Generations (ITNG), 2011 Eighth International Conference on*, 2011, p. 1025–1030

[JMUW 2012] JMUW: *Chair of Computer Science VIII: List of research projects*. 12 2012. – URL http://www8.informatik.uni-wuerzburg.de/wissenschaftforschung/

[Kim und Kim 2011] KIM, Junyeong ; KIM, Doo-Hyun: Experimental Analysis of Primary-Shadow Replication Scheme for Fault-Tolerant Operational Flight Program of Small Scale UAV. In: *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2011 14th IEEE International Symposium on*, 2011, p. 171–178. – ISSN 1555-0885

[Matsumoto et al. 2010] MATSUMOTO, K. ; UEHARA, M. ; MORI, H.: Evaluating the Fault Tolerance of Stateful TMR. In: *Network-Based Information Systems (NBiS), 2010 13th International Conference on*, 2010, p. 332–336. – ISSN 2157-0418

[Matsumoto et al. 2011] MATSUMOTO, K. ; UEHARA, M. ; MORI, H.: Evaluations of Resettable Stateful NMR. In: *Advanced Information Networking and Applications (WAINA), 2011 IEEE Workshops of International Conference on*, 2011, p. 740–745

[Montenegro 2008] MONTENEGRO, Sergio: *RODOS Introduction*. 10 2008

[Montenegro 2011] MONTENEGRO, Sergio: *RODOS IO Concept*. 01 2011

[NASA 2004] NASA: *UAV Capabilities Assessment*. 2004. – URL `http://www.uavm.com/images/NASA_UAV_Capabilities_Assessment-2004.pdf`

[Racine et al. 2002] RACINE, R. ; LEBLANC, M. ; BEILIN, S.: Design of a fault-tolerant parallel processor. In: *Digital Avionics Systems Conference, 2002. Proceedings. The 21st* Bd. 2, 2002, p. 13D2–1–13D2–10 vol.2

[Sandhana 2013] SANDHANA, Lakshmi: *MIT's SkyCall system uses a drone to guide people around campus*. 9 2013. – URL `http://www.gizmag.com/mit-skycall-guides-users-on-campus/29080/`

[Son et al. 2007] SON, Byong J. ; PARK, Ji H. ; HA, Kyung N. ; LEE, Suk ; LEE, Kyung-Chang ; KIM, Man-Ho: Development of Predictive Hybrid Redundancy for Fault tolerant in Safety Critical Systems. In: *Mechatronics, ICM2007 4th IEEE International Conference on*, 2007, p. 1–6

[Stojcsics 2012] STOJCSICS, D.: Flight safety improvements for small size unmanned aerial vehicles. In: *Intelligent Engineering Systems (INES), 2012 IEEE 16th International Conference on*, 2012, p. 483–487

[Taub 2013] TAUB, Julian: *How Future Drones Could Become Like Pets*. 3 2013. – URL `http://www.livescience.com/27984-how-future-drones-could-become-like-pets.html`

[Ulbrich et al. 2012] ULBRICH, P. ; HOFFMANN, M. ; KAPITZA, R. ; LOHMANN, D. ; SCHRODER-PREIKSCHAT, W. ; SCHMID, R.: Eliminating Single Points of Failure in Software-Based Redundancy. In: *Dependable Computing Conference (EDCC), 2012 Ninth European*, 2012, p. 49–60

[Xiaojun et al. 2012] XIAOJUN, Duan ; CHENGFU, Wu ; HUAIMIN, Chen ; JIANGXIA, Zhang: A rapid test platform of redundant flight control computer software based on RFM. In: *Computer Science Education (ICCSE), 2012 7th International Conference on*, 2012, p. 403–406

[Yang und Kwak 2010] YANG, J. M. ; KWAK, S.W.: A checkpoint scheme with task duplication considering transient and permanent faults. In: *Industrial Engineering and Engineering Management (IEEM), 2010 IEEE International Conference on*, 2010, p. 606–610. – ISSN 2157-3611

[Yellowley 2006] YELLOWLEY, Mathieu Bouvier I.: A distributed reconfigurable architecture for fault-tolerant systems. In: *International Journal of Manufacturing Technology and Management* 8 (2006), 01, p. 141–158