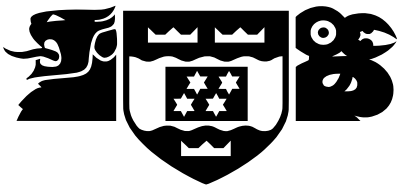
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui



School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Demonstrating Whiley on an Embedded System

Matt Stevens

Supervisor: David Pearce

Submitted in partial fulfilment of the requirements for Bachelor of Engineering with Honours in Software Engineering.

Abstract

Developing and verifying the software for safety critical embedded systems can be difficult and expensive due to unique constraints, including limited RAM and minimalist operating systems. This report looks at Whiley, a verifying compiler, which is intended to improve the correctness of code on a variety of systems. For the first time, Whiley is explored in the embedded systems context to identify obstacles to becoming a practical tool for embedded systems programmers. The conclusion identifies three areas of work for the Whiley project; resolving memory management issues inherent in embedded systems, facilitating unbounded to bounded datatype conversions and improving the ability to determine bytecode context. The forth conclusion is to adopt the use of industry debugging tools, reflecting the difficulty of debugging an embedded system. This work built a Whiley to C compiler and using it, achieved demonstrating Whiley on an embedded system—the Bitcraze Crazyflie Quad-copter. Experiments conclude that the code automatically generated by the Whiley to C compiler, performs comparably to the original C code.

Contents

Figures

Chapter 1

Introduction

> "Ubiquitous computing has as its goal the non-intrusive availability of computers throughout the physical environment, virtually, if not effectively, invisible to the user." —Mark Weiser (1993)[1]

Advances in microprocessors and supporting technologies have enabled this vision—expressed 20 years ago—of Ubiquitous Computing using embedded systems. The *Internet of Things* is a more recent concept [2] referring to embedded systems that are connected to the internet. Embedded systems today surround us, we are reliant on them controlling for example; phones, car brakes and payment systems; the Crazyflie illustrated in Figure 1.1 is another example. Tomorrow, this will extend to driver-less vehicles, connected appliances, home aid robots and more.

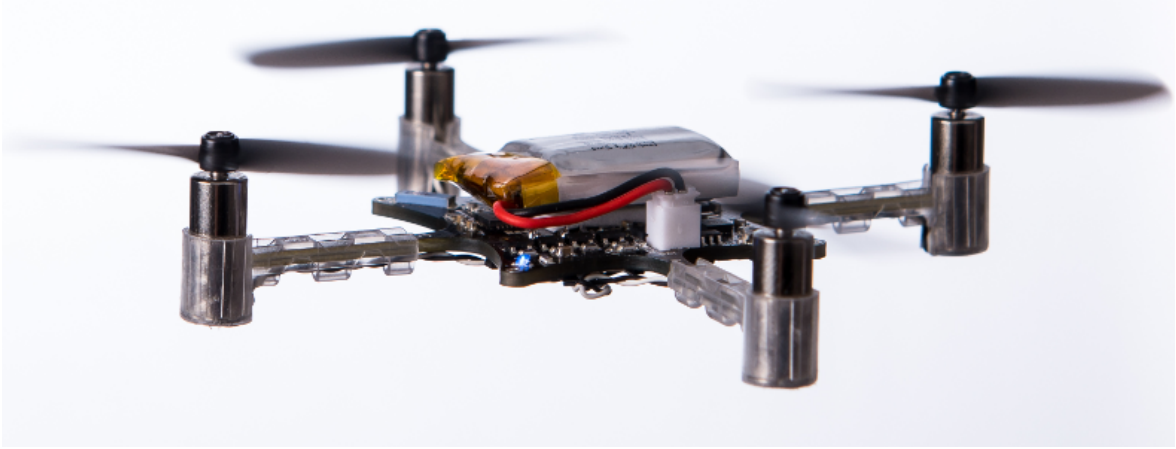


Figure 1.1: The Bitcraze Crazyflie. [1]

The number of existing embedded systems is staggering. For example, the car population topped 1 billion units in 2010 [3], averaging over 70 microprocessors each [4, 5]. These numbers are expected to increase further [2]. Many embedded systems are, and will be, managed through the *Internet of Things*, which provides a means for easily sending and receiving data, plus updating existing software. One concern is that the ability to push faulty software across millions of devices simultaneously, has potential to lead to causing havoc, destruction of property and personal injuries on a global scale.

Safety and security are key concerns. As the microprocessors controlling embedded systems get more advanced, the programs running them get more complicated and con-

[1] Bitcraze. Accessed: Oct 2014. http://www.bitcraze.se

sequently more difficult to prove they will work as intended [5, 6]. Embedded systems are expected to continue to be developed in the industrial sectors of: Aviation, Aerospace, Automotive, Telecommunications, Healthcare and Independent Living to name a few [7, 8, 9, 5] and there are numerous well studied examples of embedded system failures, such as the Toyota Motor Corporation's brakes, the Ariane 5 rocket, Therac-25 medical equipment and the Mars lander [10, 11, 12, 13].

There are ways to prove that a system works. For example, utilising Engineers to manually prove a program using mathematical modelling [14]; using tools like Event-B [15] and Rodin [16]; and using automated Model Based Testing [17, 18, 19, 20, 21] to generate exhaustive tests based on defensible strategies. Despite these, proving a system correct is time consuming and expensive, and it is generally only done for critical systems. One example is the driver-less Paris Metro system, which was modelled first using a variant of the mathematical Z notation—later becoming B notation—before code was generated from the model. This successful project lead to Event-B, but progress using Event-B in industry since then has been slow [22, 23].

Not proving a system to be correct can be costly. Poorly written code in car braking systems developed by Toyota Motor Corporation, has been a contributing factor to at least 34 deaths [24, 12, 25]. To date Toyota has been fined US$1.2 billion by the courts and a further US$1.6 billion has been awarded to class action complainants [26, 27].

Embedded systems will continue to fail. As more incidents occur, the impact on global brands will drive the search for better solutions. This project takes a step in that direction, by demonstrating Whiley, a verifying compiler, being used on an embedded system, to explore the issues it needs to address in order to become a practical tool for embedded systems programmers. In the process showing it can perform comparably with C, the main language used for programming embedded systems in industry.

1.1 Contributions

This project contributes to ongoing research in Whiley—by exploring how Whiley can be adapted to meet the demands of the embedded system environment, by creating a compiler that translates from Whiley code to C code for use on the Crazyflie quad-copter. It is hoped this research will favourably influence Whileys future development in the embedded system space. The key contributions made are:

- Designed and implemented a tool to translate Whiley code to C code, suitable for embedded systems.
- Demonstrated Whiley being used on a real embedded system, the Bitcraze Crazyflie Quad-copter.
- Conducted experiments to compare the performance of the Whiley code with the original Crazyflie code.
- Identified several issues for the Whiley project to address in order to become a practical tool for embedded systems programmers.

From here Chapter 2 discusses background material important to understanding the project, Chapter 3 discusses the design of the solution while Chapter 4 discusses the translation from Whiley to C. Chapter 5 highlights issues relating to porting C to Whiley and Whiley's integration into the original Crazyflie code. Chapter 6 evaluates the new code against the original Crazyflie code, while Chapter 7 concludes and discusses possible future work.

Chapter 2

Background

2.1 Embedded Systems

At the heart of Ubiquitous Computing and the Internet of Things are black box embedded systems which are expected to "just work", and work predictably and safely. Experiences outside this norm tends to lead to unhappy consumers and in the worst case scenarios may involve consumer death or injury [24].

The name Embedded Systems implies a system that is embedded into other systems. While this is often the case, such as car braking systems that cannot function on their own; the title also covers other stand-alone systems that merely take input or react to a timer and generate an output. Network routers are one example, pacemakers are another. An embedded system is perhaps better described as an application-specific system that involves the close co-ordination of the device, the computing hardware and the software, to facilitate turning inputs into useful outputs in a timely manner [28].

The microcontrollers in embedded systems, share common properties: limited processor power, limited RAM, limited flash memory, and a minimalist operating system that emphasises predictability and response times. The microcontroller is termed the target system, while the software is typically written on a desktop computer (the host system) in the C programming language. A cross-compiler is used on the host, to generate a binary image that is then flashed[1] to the target system.

The Crazyflie is considered a soft real-time system. It receives pilot input, processes this input in a timely manner and uses the output to individually control four motors (see Figure 2.1). The Crazyflie is an example of a real-time system that provides both functional and timely responses, without which the Crazyflie controls may be un-usable. To be a hard real-time system the Crazyflie would ensure timing deadlines are met and would consider a failed deadline as a software failure; a level of control usually reserved for safety-critical systems.

2.2 Whiley, a Verifying Compiler

The Whiley Programming Language [29, 30, 31] is being developed by Senior Lecturer Dr David Pearce at Victoria University of Wellington. The aim is to achieve a verifying compiler [32]—one of the grand challenges for computer science set by Prof. Sir Tony Hoare (ACM Turing Award winner, FRS) in 2003 [33]. One of the benefits to society of meeting Hoare's

[1] To flash a binary image, is to copy it to the devices flash memory. Flash memory is persistent, erasable, read-only memory.

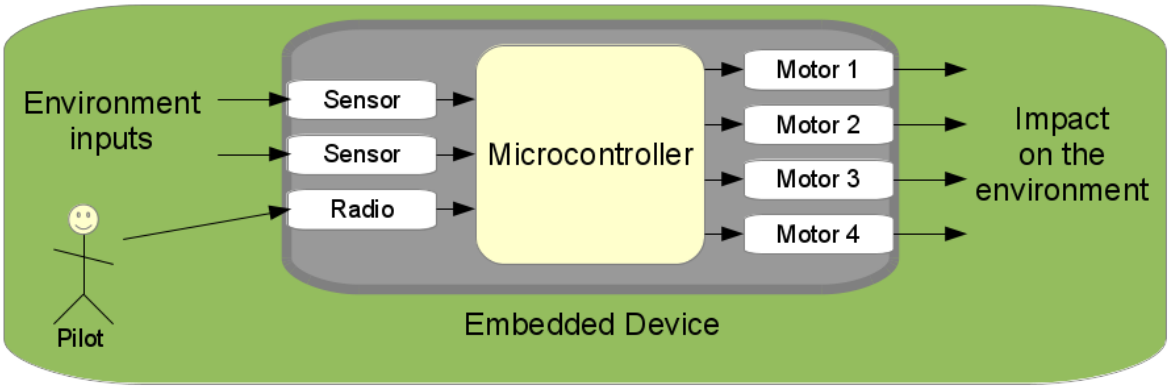


Figure 2.1: The Crazyflie pilot provides inputs (thrust, pitch, yaw and roll), via a wireless connection to the microcontroller. The microcontroller uses inputs to create outputs which direct motor speeds to influence its position in the environment.

grand challenge, is to enable complex systems like Adaptive Cruise Control [34] to be proven to be defect free by the software used to implement it.

Whiley has been designed from the ground up to facilitate the generation of mathematically verified programs. For example it uses the Functional Programming paradigm which promotes "pure" functions, functions that have no side effects. These functions are easy to reason about and may form the basic building blocks to reason over larger modules. Whiley and Functional Programming also avoid global variables for similar reasons; global state, shared across functions makes those functions impure.

2.2.1 Whiley Features

Unbounded Integers and Reals

Whiley allows the expression of very large numbers, which means that arithmetic operations are more precise and can handle values vastly greater than C based languages where arithmetic types are bounded. For example a signed **int** in a C based language on the x86_64 architecture, is 4 bytes in size and can express a value in the range of -2,147,483,648 to 2,147,483,647. In comparison, Whiley unbounded values can freely use available memory to express very large numbers. For instance a Whiley **real** can easily express pi to 100 decimal places or more.

Compound types

Whiley has a range of compound types, of which two are used in this project: records and lists. These will both feel familiar to programmers. In brief:

Records use sets of key:value pairs. e.g., {alice => 45, bob => 22}

Lists may be thought of as similar to arrays in C based languages, e.g., [1, 3, 5, 7]. However other data structures, such as linked lists, are also comparable.

Whiley lists deserve a little more elaboration as they provide translation challenges later in this report. Lists enable various high level data manipulation techniques. Two examples of this are: an intrinsic size operator that allows iterating over lists and an append operator that allows two lists to be joined dynamically at runtime [31]. These examples will be used later when discussing how to implement Whiley lists in C (see Section 4.1.2).

Verification

Whiley uses a common approach to specifying software, where programmers provide pre-conditions, post-conditions and loop invariants [35, 36, 37]. If the program does not subsequently verify, the Whiley verifier will generate an error message to the programmer highlighting the failure point. If desired, a programmer may choose to compile a Whiley program with verification turned off.

```
1  function test(int x) => (int r)
2    requires 5 <= x && x <= 10 // keyword ''requires'', specifies the pre−condition for x
3    ensures 6 <= r && r <= 11: // keyword ''ensures'', specifies the post−condition for r
4    int i = 0
5    int ghost_x = x
6    while x < 11
7      where x − i == ghost_x: // keyword ''where'', specifies the loop invariant
8      x = x + 1
9      i = i + 1
10   return x
```

Listing 2.1: Whiley function with Pre and Post-conditions and Loop Invariants.

Pre-conditions specify invariants that must be true before a function may be used. For example in Listing 2.1, the parameter x must be a value from 5 to 10 as specified by the **requires** keyword. If 6 is used as an input then the code will verify, but use 4 and the verification fails with the message "pre-condition not satisfied".

A post-condition is very similar. In Listing 2.1 it uses the **ensures** keyword and specifies the bounds of the output. In this example the return value r must fall in the range 6 to 11.

The last example in Listing 2.1 is a loop invariant which uses the **where** keyword. This checks a condition of the loop to ensures that $x - i$ always equals the original value of x (ghost_x).

Whiley's verification process leads to an interesting and desirable trait in Whiley bytecode; it is now verified as satisfying the conditions and invariants in the Whiley source code.

Bytecode

Whiley compiles to Whiley bytecode, which contains all the elements required to translate each bytecode into another format. Listings 2.2 and 2.3 show an example of Whiley code and its corresponding Whiley bytecode. The next translation might be into, for example, Java bytecode ready for the Java Virtual Machine or C code. There are over 60 Whiley bytecodes, Appendix B has details.

```
1  // Whiley binary arithmetic
2  i = x * y
```

Listing 2.2: Whiley code.

```
1  // Bytecode binary arithmetic
2  mul %9 = %5, %6 : int
```

Listing 2.3: Whiley bytecode.

Existing Whiley Test Suite

The Whiley project has an existing test suite of 610 unit tests for regression testing the Whiley project code base. The tests check datatypes, operations on datatypes, conditional statements, loops and other constructs. The test output is the result of running these tests and takes the form of output strings, a test that swaps the order of two tuples will output a string

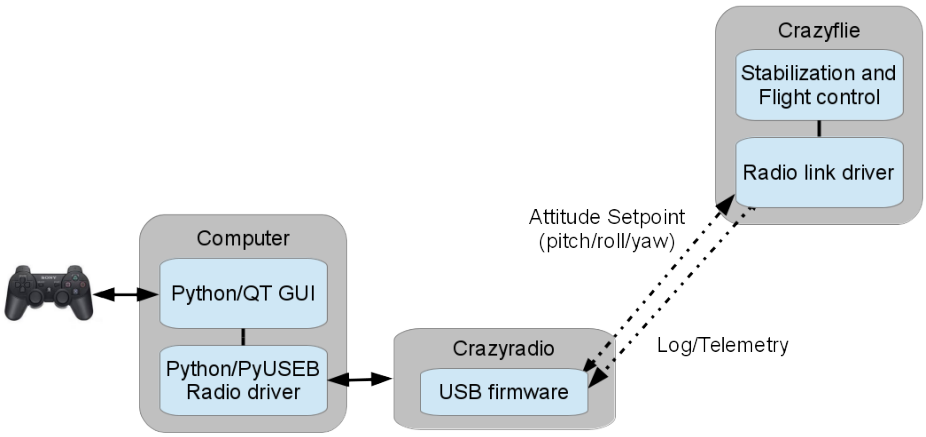


Figure 2.2: The Crazyflie system.[2]

showing the tuple in the new order. A test harness iterates through each test, setting up and tearing down the environment, including running the resulting code through the JVM, or using GCC to create executable code.

2.3 The Target Platform

The Crazyflie quad-copter [38], as shown in Figure 1.1, is the target platform for this project. It is a 19 gram quad-copter designed and sold by Bitcraze as a test platform for enthusiasts and researchers. The Crazyflie system is shown in Figure 2.2, where the pilot uses a Playstation controller, a host computer collects the pilot inputs and forwards them to the Crazyradio dongle, which wirelessly communicates with the Crazyflie itself. In addition to the radio link software, the Crazyflie quad-copter also runs flight control software, including the stabilizer algorithm.

The Crazyflie microprocessor [39] is the STM32F103CB—designed by ARM Holdings [40] and manufactured by STMicroelectronics [41]—with 20kb RAM and 120kb flash memory. The software is written in C and a GCC compiler for the STM32F103CB microprocessor is available. The Crazyflie software is open source and publicly available on Github [42].

Microprocessors that are similar to the STM32F103CB feature in many embedded systems such as cars, amplifiers, clocks, TV, medical devices, industrial tools and monitoring equipment. By demonstrating Whiley on the STM32F103CB it is anticipated that, at a later date, Whiley may be used on many other embedded devices.

2.3.1 Crazyflie Software Architecture

The software running on the Crazyflie contains a core stabilizer algorithm that keeps the Crazyflie flying level and allows it to react predictably to pilot inputs (see Figure 2.3). A review of the code showed the stabilizer algorithm consists of three modules that are self contained (stabilizer.c, controller.c and pid.c) and run within a single RTOS task (see Section 2.3.3). The effect of the stabilizer algorithm can be felt when holding the running Crazyflie; in that tilting it causes the code to attempt to level again. As the lower rotors gain power and the higher rotors lose power, this induces a feeling of resistance, similar to attempting to tilt a gyroscope.

[2]Bitcraze. Accessed Oct 2014. Retrieved from: http://wiki.bitcraze.se/projects:crazyflie:userguide:index

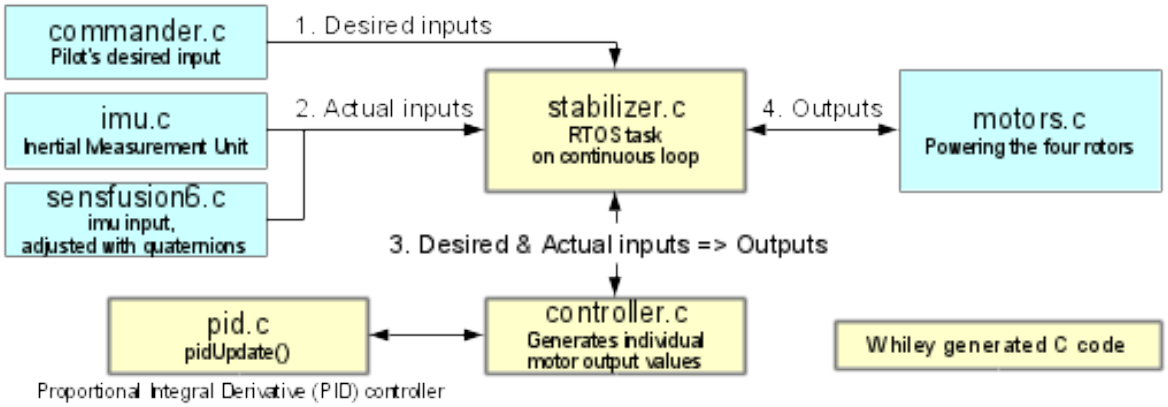


Figure 2.3: Crazyflie architecture, showing the Stabilizer algorithm in context.

The stabilizer algorithm accepts desired input from the pilot (thrust, roll, pitch and yaw), actual position input from the sensors (roll, pitch and yaw) and using a Proportional Integral Derivative (PID) controller [43], generates outputs that control individual motor speeds in a manner designed to increase stability. The existing code for the stabilizer (see Appendix D) provides a working application to emulate and the manually generated Whiley code stays close to the original algorithm (see Section 5.1).

One of the advantages of targeting the stabilizing modules is that the projects aim of demonstrating Whiley on an embedded system can be achieved through replacing a key subset of code. In addition this module may be sensitive to differences between the two code implementations, which can provide a basis for comparative tests.

2.3.2 C

Many embedded devices use C which was developed initially in the '70s [44]. The Crazyflie implementation conforms to the C89 standard [45, 46], although there are more recent standards (C99 and C11) and specialised extensions for embedded C [47].

2.3.3 The FreeRTOS Operating System

The Crazyflie uses a Real-Time Operating System (RTOS), specifically FreeRTOS [48, 49], an open source system under a modified GNU General Public Licence[3].

An RTOS can be either "hard" or "soft" with regard to deadlines. A "hard" system is where the correctness of the system depends not only on its functional correctness but also the timeliness of its outputs [28, 50]. Timeliness is determined by the environment; pacemakers have stricter needs than a television remote. Hard real-time systems guarantee meeting a deadline and consider a correct output performed late to be a bug [5]. Safety-critical systems will typically be hard real-time systems. Soft real-time systems value computations completed after the deadline has passed; a best effort approach, for example, the Crazyflie which prioritises important tasks but does not enforce hard deadlines.

The FreeRTOS operating system assigns jobs to tasks, places them in priority queues and uses a thread ticker[4] to run tasks concurrently (see Figure 2.4). FreeRTOS can be thought of as a thread controller that guarantees a predictable response in a small memory footprint

[3]The modification is to allow the use of FreeRTOS in proprietary software without requiring the entire application to be open source.

[4]This enables pre-emptive time slicing. A thread ticker controls each tasks access to the cpu by swapping them in and out very fast, which gives the appearance of concurrency.

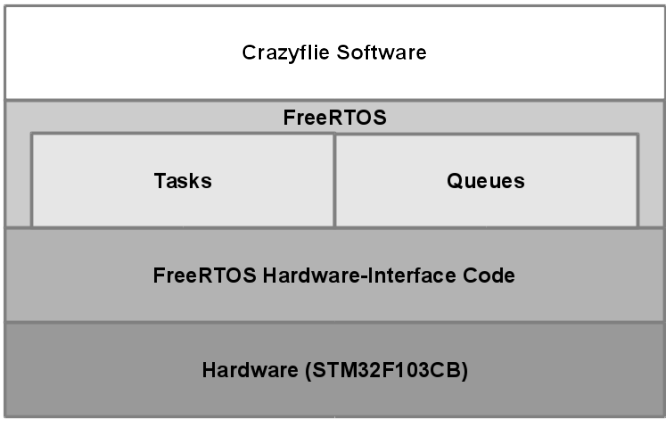


Figure 2.4: The FreeRTOS stack

of approximately 5-10 kilobytes [51]. This can be compared to Windows CE, also a RTOS, which requires approximately a megabyte of memory, to support substantially more features.

2.3.4 Memory

Memory is a scarce resource in many embedded systems. A desktop computer has gigabytes of RAM, the Crazyflie in comparison has 20 kilobytes. How memory is organised and used becomes important when it is scarce.

CPU memory, whether desktop or embedded system, is typically arranged in several memory blocks (see Figure 2.5) [52]. The stack which starts at the top and grows down, the program text and data segment at the bottom occupying a fixed space, and the heap which starts from the top of the data segment and grows up. Between the stack and the heap is the available free memory.

Methods use stack memory to create a stack frame to hold method variables and a pointer to the return point in the parent method. Variable quantity and variable size impact the size of the stack. Heap memory holds global values and values related to task scheduling, such as mutexs and semaphores. The data segment holds read-only static values and literals, such as string literals.

Heap memory is the most flexible of the three memory types—its memory is allocated at runtime rather than determined at compile time. This means for example:

- Data structures may be expanded easily when the data size is only known at run-time.
- Short term storage of large data blocks may be better handled on the heap, as it allows the easy release of that memory when done. In comparison, memory allocated on the stack is reserved for the life of the method, not just when it is needed.
- The data stored on the heap is not subject to method scope considerations. For example a Whiley list uses the heap and is independent of any method and its stack frame.

There are a variety of memory architectures, ranging from energy efficient scratch-pad memory, used to keep frequently referenced variables and instructions within a small memory space [53], to real-time operating systems like FreeRTOS that provide a range of memory strategies to choose from[5]. Such architectures do not use POSIX[6] standards[54] for allocating memory, they use specialised implementations of malloc() which can catch the unwary.

[5]FreeRTOS has, at the time of writing, 5 heap memory management options.
[6]Portable Operating System Interface

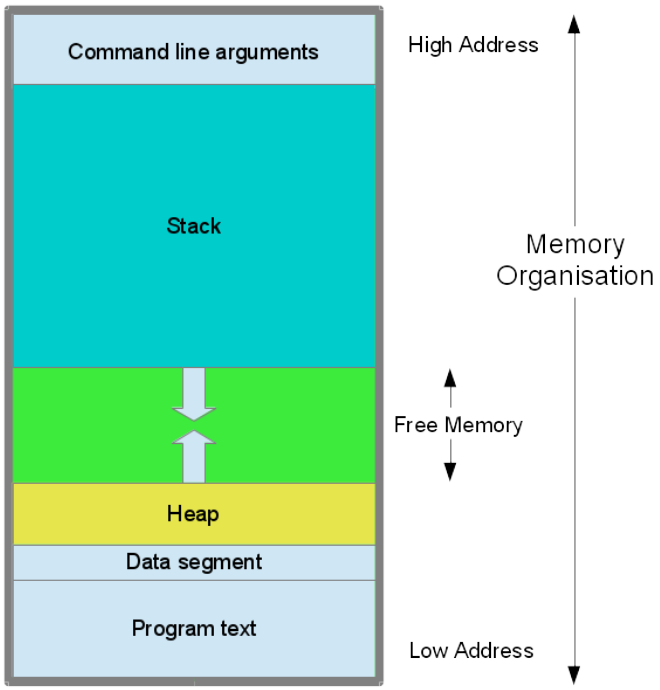


Figure 2.5: The Crazyflie's 20KB Memory

If malloc() is inadvertently used, this leads to establishing a second heap of 64 kilobytes [55] on a device which may not have this memory available.

This chapter discussed embedded systems, some of the features of Whiley and the Crazyflie embedded system—its software architecture, its FreeRTOS operating system and an overview of dynamic memory. Armed with this we move next to discussing design considerations.

Chapter 3

Design

The motivation for this work is to demonstrate using Whiley instead of C to program embedded devices. When Whiley is industry ready, this will potentially enable lower cost program verification which may lower the time and cost barriers to industrial verification. The project aim is to:

Demonstrate Whiley on an embedded system.

This will be achieved by replacing existing C code on an embedded system with equivalent Whiley generated code. The process for this is illustrated in Figure 3.1. Whiley code is first written and perhaps verified. It is then compiled to Whiley bytecode before the C file is created using a Whiley to C compiler. Finally the GCC complier creates the binary image ready to be flashed to the embedded system.

3.1 Overview

The goal is to fly the Crazyflie with new code written in Whiley—ideally retaining the same functionality and performance as the original unmodified Crazyflie code.

To make this a reality there are two steps that need to be taken in order to create the initial Whiley code—porting and integration. These steps enable targeting a portion of the original Crazyflie code and also enable evaluating the new code, using the original Crazyflie code as a benchmark. Once the Whiley code is created and integrated, the translation in Figure 3.1 can be done. The steps are:

1. **Port** the original Crazyflie stabilizer code to Whiley code.
2. **Integrate** the new Whiley stabilizer code with the original Crazyflie code.
3. **Translate** using the new Whiley to C compiler to automatically generate C code for the stabilizer.

The most important of these steps is the third step, the translation from Whiley to C; particularly as it required creating the Whiley to C compiler artefact. However the first and second steps provide the Whiley code to translate.

The rest of this chapter outlines several design considerations: discussing the implications of the Crazyflies memory constraints, the transliteration process from Whiley to C and other options for compilation targets.

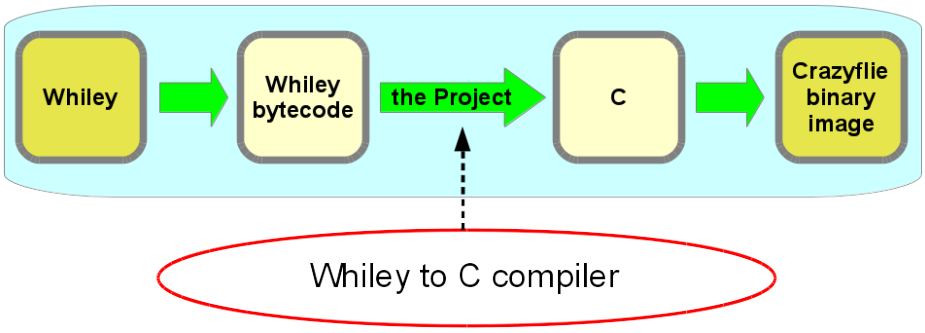


Figure 3.1: The context for the Whiley to C compiler.

3.2 Targeting the CrazyFlie

As an embedded system the Crazyflie has constraints in the form of processing power and available RAM memory. It uses a specialised RTOS tailored for these constraints, designed to minimise memory footprint and run tasks reliably. The way in which the Crazyflies software is written is also somewhat specialised, both in the way it interfaces with the RTOS and the care programmers take in managing memory. This has implications for an existing Whiley to C compiler, discussed next and how dynamic memory is handled.

3.2.1 The Existing Whiley to C Compiler

Memory constraints and minimalist operating systems are the two main reasons why an existing Whiley to C compiler, developed in 2013 [56], is not suited to embedded devices. It was written with desktop computers in mind with gigabytes of memory and a fully featured operating system. It was not intended to meet the constraints seen on embedded devices, in particular because it allocates heap memory in an unrestricted fashion, facilitated by memory management systems typical in fully featured operating systems.

3.2.2 Dynamic Memory

There are good reasons why embedded systems programmers choose to use dynamic memory. For example, a common use of heap memory is to store very large data elements. This can aid memory efficiency by enabling runtime memory allocation and memory recovery—independent of the method stack frame's life cycle—which enables efficient and significant memory re-use, both within and between methods.

But there is also a need to be conservative with memory, for example when there is only 20 kilobytes to work with. While dynamic memory offers perhaps the most flexible memory option—by allowing programmers the ability to allocate, resize and recover memory on demand at runtime—its Achilles heal is that the memory must be released when it is no longer needed, and not doing so will inevitably result in the software crashing due to an "out of memory" error. With manually crafted code, managing memory to avoid out of memory errors is relatively straightforward for the programmer, but if a mistake is made, finding memory leaks can be an onerous task.

Memory management using an algorithm is an ongoing research area. There are existing solutions such as Java's garbage collection, but these are too resource-hungry for many embedded systems. Alternative approaches include for example, Escape Analysis, which uses a pre-compilation step to examine pointers with the goal of determining scope—this can then be used to assist with considering whether the memory associated with the pointer

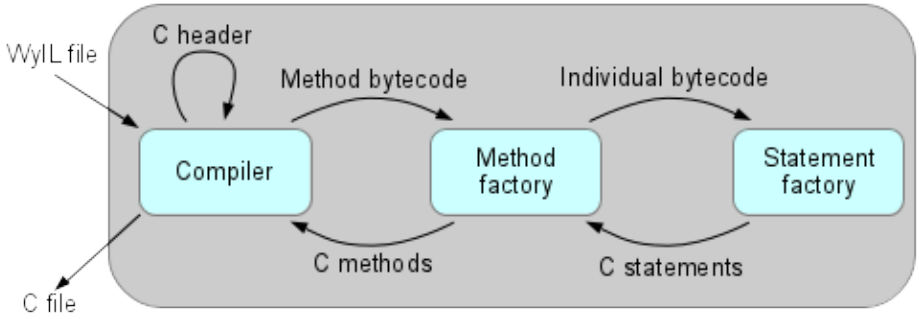


Figure 3.2: Whiley to C compiler, state machine.

should be heap or stack allocated. A range of approaches is required to achieve automated memory management and to attain this for embedded systems, it must operate in a small and efficient package. This is a big challenge and one that is left for other researchers.

Consequently an alternative solution was sought, one that avoided the need for releasing memory by minimising or eliminating the use of dynamic memory. One strategy used is refactoring to eliminate heap use, which is discussed further in Section 5.1 on porting the original Crazyflie code from C to Whiley.

3.3 Transliteration of Whiley to C

There are several approaches that may be taken when creating a compiler, Waters (1988) describes two [57].

- Transliteration followed by refinement
- Abstraction followed by reimplementation

Transliteration is the literal translation of the source code, line by line. It is a strategy which enables the translation to quickly achieve the core translation task. The refinement step then deals with any complicated issues, such as identifying blocks of code that may benefit from refactoring. The weakness of the transliteration approach is that because each element is addressed in isolation, the element's context is not taken into account. This can lead to difficulties in interpretation.

Abstraction creates an abstract model of the source software (e.g. an abstract syntax tree) and uses this model to supply context when re-implementing the code in a new language. This mitigates the primary weakness of the transliteration approach, but at the cost of increasing the complexity of the translation task.

The Whiley to C compiler uses transliteration as Waters (1988) considered it the most pragmatic approach when compiling to a lower level language. However the difficulty of interpreting bytecodes in isolation will come up in Chapter 4 when discussing transliteration in practise.

3.3.1 Whiley to C Compiler's Architecture

The Whiley to C compiler can be described as a program transformation—an automated process that takes one program and generates another [58, 59, 60, 61]. To achieve this (see Figure 3.2), the Whiley to C compiler consists of a core algorithm that accepts a Whiley bytecode (WyIL) file and generates header information and method signatures. It then iterates

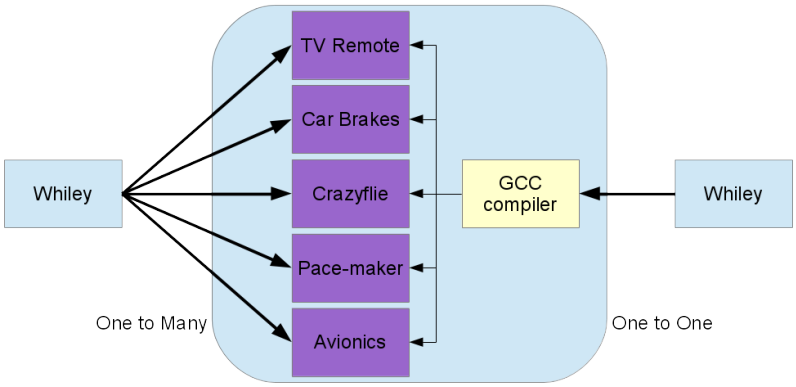


Figure 3.3: Whiley compiles to many targets (left), or Whiley compiles to one (right).

through the methods held by the WyIL file, passing each to a method factory, which passes individual bytecodes to a statement factory. The resulting text is then aggregated and output as a C file.

3.3.2 Code Generation

Several targets other than C were considered for the Whiley to C compiler. Other choices were:

- **Compiling directly to the microprocessor.** This involves translating from Whiley bytecode to machine readable binary. However this means a compiler for every make of microprocessor is required. This is shown on the left side of Figure 3.3, as a one to many relationship. While the right side utilizes the GCC collection[1] [62] as an intermediary, greatly simplifying the Whiley compiler task to interfacing with only a GCC compatible language.
- **Compiling from Whiley bytecode to something other than C.** The GCC collection compiles from a variety of languages, including Java, C, C++ and others. Java was briefly considered, however C allows direct memory management and is an industry standard for programming embedded devices; it is also the language the Crazyflie is programmed in and the translated code must integrate and compile with C. The choice of C was therefore driven by industry standards, the GCC collection and the original Crazyflie code; all preferring the C language.

In this chapter we have considered the main aspects of the project from a high-level. The next chapter will explore these in more detail.

[1] the GCC collection already undertakes to create and maintain compilers to most commercially available microchips.

Chapter 4

Transliteration

The Whiley to C compiler created as part of this project, uses the transliteration process (see Section 3.3), which proved to be straightforward to implement, but raised numerous issues that had to be resolved. This chapter discusses the detail of the transliteration process and the issues found, this includes the translation of; Whiley datatypes to C datatypes and Whiley bytecode to C statements—both of which are impacted by constraints in embedded systems. Section 4.3 discusses testing the Whiley to C compiler against the Whiley project test suite and Section 4.4 discusses the difficult and time-consuming task of debugging on an embedded system.

4.1 Data Types

For pragmatic reasons, only a subset of Whiley datatypes were translated—those needed to demonstrate Whiley on the Crazyflie, which are detailed in Figures 4.1 and 4.2.

There are multiple ways to represent a Whiley datatype as a C datatype, and individual Whiley bytecodes (see Section 4.2) may not always provide sufficient context to determine the appropriate choice. In these cases a compromise was sought. For example a string can be implemented three ways in the C programming language[1], with the choice dependant on which memory location is preferred: the data segment, the heap or the stack (see Section 2.3.4).

A complicating factor is that without context, every string in a bytecode appears the same and will therefore receive the same translation—which may have far reaching implications. For example it may impact on the verified property of the Whiley bytecode (see Section 2.2).

The following sections discuss primitive datatypes first, followed by compound types.

4.1.1 Primitive Types

Whiley represents integers and floating point numbers as unbounded (see Section 2.2.1) and Whiley chars as Unicode values of up to four bytes in size. This brings significant advantages to the precision of calculations and variety of languages supported, but at the cost of memory use. In contrast C datatypes are bound to a fixed memory size, which is a trait favoured by programmers of systems with severe memory constraints.

As shown in Figure 4.1 most of Whiley's primitive types have a corresponding C primitive type. These are:

[1]String literals are held in the data segment as read only data, string arrays in the stack are not preserved after the method completes, strings using malloc() on the heap must eventually have the memory released.

Whiley	C	Notes
char	char	Whiley char accepts Unicode up to 4 bytes in size. C char is limited to 1 byte.
int	int	Whiley int's are unbounded. C types are bounded meaning a loss of precision, but more efficient memory use.
real	float	Whiley real's are unbounded. C types are bounded, meaning a loss of precision, but more efficient memory use.
boolean	bool	C requires the stdbool.h library to recognise booleans
any	n/a	Whiley any is the supertype of all other datatypes. C has no direct equivalent.

Figure 4.1: Whiley primitive types, how they translate to C.

- **Char.** The original Crazyflie code only uses ASCII which translates readily to C chars and avoided the problems that would have been seen trying to translate multi-byte UTF-8 character encoding into one byte, where the loss of information in the process may have corrupted the data.

- **Booleans.** Booleans are not supported natively in C and required importing the standard boolean library for C, but otherwise posed no problems.

- **Arithmetic values.** Whiley unbounded **int** and **real** are translated to C unbounded **int** and **float**.

The translation of Whiley unbounded arithmetic values to C bounded values, has several ramifications: there is potential loss of precision, potential corruption of data and potentially any verified status the Whiley bytecode may have had is compromised. There is also the potential for very difficult to find bugs. This means care would need to be taken if using this system outside the scope of this project.

Despite these problems, a solution was required that met the goal of demonstrating Whiley on the Crazyflie. Consideration was given to translating an unbounded **int** to the largest integer C can support, a **long long**—which on a x86_64 machine is 8 bytes in size (the same as a **long**). However this is too small to satisfy Whiley's unbounded **int**, yet unnecessarily large for the original Crazyflie code—consequently the smaller **int** and **float** were chosen and used, without any observable ill-effect.

One of the drawbacks of the transliteration process discussed in Section 3.3, is that bytecodes are translated without context. This is apparent when considering the translation of arithmetic values; without context every Whiley **int** appears the same and will receive the same treatment, a translation to a C **int**. The Crazyflie only needed C **int** and **float**, however if a **long** was required, every Whiley **int** would have had to be translated to be a **long**.

4.1.2 Compound Types

Three compound types were used and are shown in Figure 4.2. Others exist, such as tuples, sets and maps, but were not needed for this project.

Of the three compound types, Whiley records posed the least problems. They are collections of name:value pairs, similar to dictionaries in Python. The translation to a C struct is straight forward as can be seen in Listings 4.1 and 4.2.

Whiley	C	Notes
string	char*	Strings in C may use stack, heap or read-only data memory, using respectively; arrays, malloc() or string literals.
[T]	T[ ]	Whiley Lists are translated to an array. Lists carry length information, arrays in C do not, requiring care when passing them out of scope. Lists are stored in heap memory, arrays (in this compiler—see text) are stored on the stack.
{T x, T y}	Struct { T x; T y; }	Whiley Records translates to a C struct.

Figure 4.2: Whiley compound types, how they translate to C.

```
1  // a Whiley record
2  type PidObject is {
3      real desired,
4      real error,
5      real prevError,
6      real integ
7  }
```

Listing 4.1: A Whiley Record.

```
1  // a C struct
2  typedef struct {
3      float desired;
4      float error;
5      float prevError;
6      float integ;
7  } PidObject;
```

Listing 4.2: Becomes a C Struct.

The remaining two, Whiley strings and lists, posed difficulties as C programmers have several ways to implement them. The obvious way is perhaps to translate them to C arrays, however C arrays do not hold a size metric and have interesting storage implications—both of which are discussed in the next two sections.

Lists

Whiley lists can be represented in C, in a variety of ways. C arrays are natively supported while other solutions can be sourced from the GLib library and include linked lists, queues, sequences, hash tables and others [63]. To pick one example from GLib; a double linked list retains size information, accommodates appending, prepending and inserting plus other functions that are similar to those found in Whiley lists. However this functionality comes with a higher memory footprint—the full library is 36 megabytes—and GLib uses dynamic memory (heap memory) in many cases.

Representation

Whiley lists are perhaps most easily represented using arrays in C, they are memory efficient, fast, easy to use and can use either heap or stack memory. However Whiley lists have a length operator, while arrays do not, meaning a Whiley list can be passed to a child method and iterated over. To achieve the same, a C array must be passed to a child method where the array's length is either: inherently understood (i.e. fixed), or provided when the array is passed. It causes other problems as well, such as making it difficult to program defensively to prevent over-writing adjacent data. With these shortcoming in mind, arrays were selected to represent Whiley lists for being memory efficient, fast to implement and intuitive.

C programmers have (at least) five ways to provide size with an array:

1. Ensure child methods have a fixed, known size for the array.
2. Pass the array and the array size as a pair of method parameters.
3. Pass a struct that contains the array and a size value.

4. Reserve the first position of the array for its size.

5. End the array with a marker, similar to string arrays being null terminated.

The first was chosen for this project as it was the most intuitive to implement—there was only one instance of an array in the stabilizer code being translated—but in hindsight this choice had unnecessarily negative consequences for the number of tests passed in the Whiley test suite, using a struct holding the array and length may have been a better choice.

Storage
Whiley lists are best represented dynamically in heap memory. This allows them to be independent of method scope and able to adjust the amount of memory required at runtime. For example appending two lists together when one or both of the list's sizes cannot be determined at compile time. Another important factor is that Whiley enables the use of the heap by utilizing an algorithm to free up memory—similar to Java's garbage collector—which is typically not an option for embedded systems (see Section 3.2.2).

```
1 // Whiley list
2 [int] list = new [1, 2, 3]
```

Listing 4.3: A Whiley List.

```
1 // C array declaration on the stack
2 int list[ ] = {1, 2, 3};
3
4 // C array declaration on the heap
5 int *list = malloc(3 * sizeof(int));
6     list[0] = 1;
7     list[1] = 2;
8     list[3] = 3;
```

Listing 4.4: Ways to represent a C Array.

Whiley lists when translated to C arrays, have a choice. Listings 4.3 and 4.4 show the two ways a Whiley list can be implemented in C to occupy either heap or stack memory:

Heap memory allows the array to persist after the initialising method has finished, and it allows resizing. However heap memory must be specifically allocated and specifically released when no longer needed. If memory is not released, further allocations of heap memory will build up over time until all available memory is allocated and the program crashes with an out-of-memory error, also known as a memory leak.

Stack memory is tied to its enclosing method, for the life of the method, after which the values are not preserved and the memory is available to other processes. Once the method finishes, further references to the value will cause errors as the code attempts to access memory that has potentially been allocated to another method. This means the array may be passed to and used by child methods, but not passed out of the parent method as it finishes.

Using heap memory to implement Whiley lists would provide a simple generic solution. However releasing allocated memory in an embedded environment which does not have a garbage collector, can be a difficult task. In addition the original Crazyflie code avoids dynamic memory allocation for temporary data. These factors helped shape the decision to use stack memory for C arrays. Similar to the decision on translating ints and reals, this decision applied to translating all Whiley lists and could impact negatively on any verified status the bytecode may have had.

Strings

Strings are represented in C as null terminated char arrays. This resolves the size problem as it makes counting the elements up to the null terminator feasible.

The choice of memory location that strings can use is similar to arrays and includes heap and stack. Strings may also use the data segment—which sits between program text and heap memory (see Figure 2.5) and is generally treated as read-only memory. Static variables are stored in this space, for example constants or literals. This gives strings the option of being treated as a string literal, which provides global scope, but no ability to change the contents of the string. Listings 4.5 and 4.6 show the three ways a Whiley string may be implemented in C.

```
1 // Whiley string
2 string str = ''Hello''
```

Listing 4.5: A Whiley string.

```
1 // C string literal − storage: data segment
2 char *str = ''Hello'';
3
4 // C string array − storage: stack
5 char str[] = ''Hello'';
6
7 // C string array − storage: heap
8 char *str = malloc( 6 ∗ sizeof(char));
9 strcpy(str, ''Hello'');
```

Listing 4.6: Ways to implement in C.

The original Crazyflie code only uses string literals as tokens to aid debugging efforts. This was ported to Whiley as string literals and subsequently strings in the Whiley to C compiler are translated into C as string literals.

4.2 Bytecode Translation

Bytecode translation, per the discussion on Transliteration (see Section 3.3), means each bytecode in a WyIL file is examined in isolation and then translated into one or more C statements. There is a list of 60 plus Whiley bytecodes is in Appendix B. Over half of the bytecodes were translated for this project, prioritising those needed for the Crazyflie application.

Whiley bytecode uses unique numeric registers rather than variable names, with each method in the WyIL file starting a new set of registers. The Whiley to C compiler uses hashmaps to map registers with variable names, for example register 22 maps to the string a22 by pre-pending the character a to the register number. The mapping is done when the register is first seen in the method and the variable is usually also initialised in the output C code.

From here simple bytecodes are discussed first, with several examples, followed by branching bytecodes that allow conditional branching such as if and loop statements. Finally two other bytecodes are described, the invoke and new bytecodes.

4.2.1 Simple Bytecodes

Listings 4.7, 4.8 and 4.9 show a trivial example of the transition from Whiley to Whiley bytecode, then to C. Figure 3.1 shows this transition, plus a further transition to binary.

```
1 // Whiley initialise int
2 int x = 5
```

Listing 4.7: Whiley code.

```
1 // Bytecode initialise int
2 const %1 = 5 : int
```

Listing 4.8: Bytecode.

```
1 // C initialise int
2 int a1 = 5;
```

Listing 4.9: C code.

On initialising a variable it's register:variable mapping is placed in the hashmap of registers.

Return bytecode

The Return bytecode is a second simple example (see Listings 4.10, 4.11 and 4.12). The Whiley to C compiler translates the return bytecode (which contains an optional return register), into a C return statement, with a variable retrieved from the hashmap of registers.

```
1 // Whiley return
2 return b
```

Listing 4.10: Whiley code.

```
1 // Bytecode return
2 return %17 : int
```

Listing 4.11: Bytecode.

```
1 // C return
2 return a17;
```

Listing 4.12: C code.

Assign bytecode

Assignment bytecodes involve assigning the value held by one register to a target register (see Listings 4.13, 4.14 and 4.15). If the target register is already in the method's hashmap of registers, then the C statement is merely `a7 = a3;`. If the target register is not in the hashmap of registers, meaning it has not yet been seen or initialised in the C code output, then the C statement is **int** `a7 = a3;` and also added to the hashmap of registers.

```
1 // Whiley assign
2 a = b
```

Listing 4.13: Whiley code.

```
1 // Bytecode assign
2 assign %7 = %3 : int
```

Listing 4.14: Bytecode.

```
1 // C assign
2 a7 = a3;
```

Listing 4.15: C code.

On occasion there are extra assign bytecodes in the Whiley bytecode. Listings 4.20 (lines 3, 8) and 4.23 (lines 3, 5, 9, 12) have examples. The Whiley to C algorithm, while resolving memory bugs, was refactored to strip out the extra assignments as they cause unnecessary variable initialisations. This was done by utilizing the hashmap of registers to map the target register to its assignee. For example in Listing 4.20 the bytecode `assign %1 = %2 :` **int** has the register `1` mapped to the variable `a2` as can be seen later in Listing 4.21 (line 7).

Binary Arithmetic bytecode

Binary arithmetic operations involve three registers, a target, left-hand-side and right-hand-side, plus an operation from the set of {+, -, *, /, %}. An example can be seen in Listings 4.16, 4.17 and 4.18. In the same manner as the Assign bytecode, a check is made to see if the target register already exists in the hashmap. If yes it is rendered as `a29 = a0 + a1;` otherwise it is added to the hashmap and rendered as **int** `a29 = a0 + a1;`.

```
1 // Whiley binary arithmetic
2 i = x + y
```

Listing 4.16: Whiley code.

```
1 // Bytecode binary arithmetic
2 add %29 = %0, %1 : int
```

Listing 4.17: Bytecode.

```
1 // C binary arithmetic
2 int a29 = a0 + a1;
```

Listing 4.18: C code.

4.2.2 Branching Bytecodes

Branching bytecodes provide conditional branching to the code, such as **if** statements. The Whiley compiler facilitates this by de-constructing complex branching structures into Whiley bytecode that uses labels and **goto** instructions for unstructured control flow. This makes the translation into C appear to copy the bytecode rather than the original Whiley instructions.

If

The Whiley **if** bytecode is the first example using conditional statements. The boolean condition in the bytecode is reversed by the Whiley WyC compiler and uses the **goto** to jump past the required action, to the next label, as can be seen in Listings 4.19, 4.20 and 4.21.

```
1 method doStuff(int x) => int:
2   int r = 0
3   if( x < 5 ):
4     r = -1
5   return r
```

Listing 4.19: Whiley code.

```
1  private int doStuff(int):
2    const %2 = 0 : int
3    assign %1 = %2 : int
4    const %4 = 5 : int
5    ifge %0, %4 goto label0 : int
6      const %5 = 1 : int
7      neg %6 = %5 : int
8      assign %1 = %6 : int
9    .label0
10   return %1 : int
```

Listing 4.20: Bytecode.

```
1  int doStuff ( int a0 ){
2    int a2 = 0;
3    int a4 = 5;
4    if( a0 >= a4 ){goto label0;};
5      int a5 = 1;
6      int a6 = -a5;
7      a2 = a6;
8    label0: ;
9    return a2;
10 }
```

Listing 4.21: C code.

Loop

Loops in Whiley are represented by three bytecodes. For example the Whiley **while** statement in Listing 4.22 is represented in bytecode in Listing 4.23, starting from line 6. The loop exit is marked by an end bytecode, on line13. These three bytecode are in turn represented by four lines of C code. In Listing 4.24, lines 4 and 5, plus lines 11 and 12.

```
1 method doStuff(int x) => int:
2   int i = 0
3   int r = 0
4   while( i < x ):
5     r = r + i
6     i = i + 1
7   return r
```

Listing 4.22: Whiley code.

```
1  private int doStuff(int):
2   const %2 = 0 : int
3   assign %1 = %2 : int
4   const %4 = 0 : int
5   assign %3 = %4 : int
6   loop (%1, %3)
7    ifge %1, %0 goto label0 : int
8     add %9 = %3, %1 : int
9     assign %3 = %9 : int
10    const %11 = 1 : int
11    add %12 = %1, %11 : int
12    assign %1 = %12 : int
13  end label0
14  return %3 : int
```

Listing 4.23: Bytecode.

```
1  int doStuff ( int a0 ){
2    int a2 = 0;
3    int a4 = 0;
4    loop_start_label0: ;
5     if(a2 >= a0){goto label0;};
6      int a9 = a4 + a2;
7      a4 = a9;
8      int a11 = 1;
9      int a12 = a2 + a11;
10     a2 = a12;
11    goto loop_start_label0;
12   label0: ;
13   return a4;
14 }
```

Listing 4.24: C code.

4.2.3 Other Bytecodes

Invoke

The Invoke bytecode is called to use an existing function. This occurs when using a method declared elsewhere in the program file, as illustrated in the main method of Listing 4.25. Listing 4.26 on line 8 and 9 shows the invoke bytecode calling the method and assigning the value returned to register 4. Listing 4.27 on line 9 shows the translated C code.

```
1 function f() => string:
2   return ''Hello World''
3
4 method main( ... ) => void:
5   string a = f()
6   ...
```

Listing 4.25: Whiley code

```
1 private string f():
2   const %0 = ''Hello World'':string
3   return %0 : string
4
5 private void main( ... ):
6   invoke %4 = () test:f :
7                 function() => string
8   ...
9   return
```

Listing 4.26: Bytecode.

```
1  char *f(void){
2    char *a0 = ''Hello World'';
3    return a0;
4  }
5
6  int main (){
7    char *a4 = f();
8    ...
9    return 0;
10 }
```

Listing 4.27: C code.

Whiley keyword - new

The Whiley keyword **new** enables a new object or collection to be created in heap memory, anticipating they are cleaned up (for example by a garbage collector) when they are no longer needed. This works well on a desktop computer, but on a device with only 20 kilobytes of RAM, memory recovery strategies like garbage collection are too expensive. Section 4.1.2 has already discussed Whiley lists and placing them on the stack rather than the heap to avoid memory management problems. For the same reasons, the Whiley **new** keyword is re-interpreted to create objects on the stack.

This re-interpretation of **new** has interesting ramifications in Whiley. The objects created are now limited to method scope and they cannot use functionality that requires them to expand in size dynamically at runtime. This reduces the range of tasks Whiley objects can do and is a limitation of this approach.

Using the re-interpreted **new**, enabled the following two types of refactoring of the original Crazyflie code:

- Refactor the original C code to move object file scope declarations into method scope. This is discussed in Section 5.1.1.
- Refactor the original C code so where objects are used in a parent function, the object initialization is also done in that parent. This mitigates one of the weaknesses of stack memory being limited to the scope of the initialising method and child methods (see Section 4.1.2). In practise this did not prove to be difficult, however it may not prove to be practical in the general case.

4.3 Testing the Whiley to C Compiler

Regression tests were used as the code base was first developed, initially testing the syntax of the output C file. This proved to be sensitive to inconsequential changes in C code output, such as formatting changes. A better solution was sought which led to the Whiley project test suite, which tests the behaviour of the output C code, rather than the syntax.

The Whiley project has a suite of 610 tests. Each test consists of a Whiley program and an oracle output (see Listings 4.28 and 4.29), enabling the output of the Whiley program to be compared against the oracle output. This framework was customised for the project and enabled the test suite to be run on the Whiley to C compiler, helping to gain confidence in the functionality that had been implemented.

```
method main(System.Console sys) => void:
    int i = 0
    while i < 5:
        if i == 3:
            break
        i = i + 1
    sys.out.println(i)
```

Listing 4.28: Test While_Valid_17.

```
3
```

Listing 4.29: Oracle output.

Of the 610 tests, 114 tests pass. The failures were primarily because they involved bytecode functionality that was not required or implemented for the project, or instances where a datatype was used which clashed with test expectations, for example Whiley lists are expected to be iterated over, C arrays were implemented in a way that did not allow for this. Passing tests were considered as a support for the project, not to be mistaken as a goal in itself.

Appendix C contains a brief overview of the Whiley test suite and why failing tests failed.

4.4 Debugging

Debugging on the Crazyflie proved to be a challenging and time-consuming part of the project. It was a difficult debugging environment, with no VDU for feedback and no room for a debugging environment. This eliminates most options used in a desktop operating system including stepping through code, reading runtime variable values, checking conditionals or running unit tests. The concern at every step was how to find and diagnose problems in the code; with the Crazyflie offering only two ways of communicating with the outside world, flashing its LEDs and spinning its rotors.

When the new Crazyflie code failed, LEDs proved helpful in finding the cause of several critical bugs, but while they could identify where the code was failing, they could not identify why; nor provide details on what the run-time variables were. It took many experiments to debug this way and consumed a lot of time.

Once several critical memory issues were resolved, tasks within the application started to compete for the use of the LEDs, making them unreliable for debugging. This left observing the rotors as the primary feedback tool.

JTAG ICE Debugger

Not wishing to rely on flashing LEDs and spinning rotors, motivated the enquiry into alternative debugging solutions. JTAG (Joint Test Action Group) is a boundary scanning architecture specified in IEEE 1149.1 which can be combined with hardware that integrates with the chip, an In-Circuit Emulator (ICE), to create a debugging tool. The JTAG unit interfaces using a set of pins mapped per IEEE 1149.1 and executes code in the target system while also allowing stepping through the code, observing variables and setting breakpoints [64, 65].

A Segger J-Link EDU JTAG unit was kindly provided by Victoria (see Figure 4.3). However, despite the potential of the JTAG unit to provide debugging capabilities (similar to

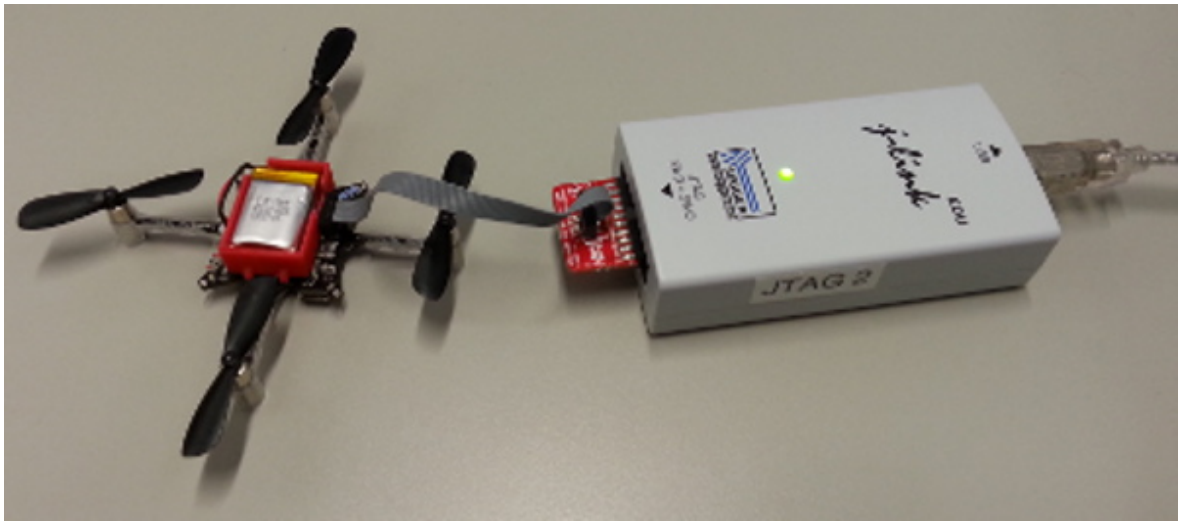


Figure 4.3: Crazyflie and JTAG.

GNU GDB), after several days without success at getting it working, debugging had to fall back to relying on flashing LEDs and spinning rotors. The lack of in-house experience with using this tool at Victoria, was unfortunate. The on-line documentation similarly was of little help, providing step-by-step pictorial instructions for Windows, but only a command line instruction for Linux.

Despite the issues raised in this chapter, the very first compiler success was achieved with translating a simple Whiley program that enabled the LEDs to flash and motors to spin. This first success was followed by re-examining the original Crazyflie code with a view to creating a Whiley version of the stabilizer algorithm. The next chapter discusses the issues faced when porting a host module to Whiley and integrating the resulting Whiley code with the remaining host modules.

Chapter 5

Porting and Integration

To demonstrate Whiley on the Crazyflie, Whiley code had to be created from the original Crazyflie stabilizer module (see Section 2.3.1). The process for this was to manually translate the stabilizer code to Whiley (porting), then ensure the new Whiley code can interface with the rest of the Crazyflie code (integrating). Once completed the Whiley code will become input for the Whiley to C compiler (see Chapter 4). The next sections discuss porting and integrating.

5.1 Porting

Porting and integration is the process of adapting software for use in a different environment, in this case C to Whiley. For the most part, this was straight forward, porting involved creating the new Whiley code, integration follows from porting and is discussed in the next section. Porting involves six stages:

1. Revise the source C file and remove redundant code. This includes analysing preprocessor commands such as #ifdef statements to identify redundant code, removing any log or test code and any functionality not required.
2. Create a reference table of the datatypes used in the C file, as this will become useful throughout the porting process.. Search the application code for definitions.
3. Identify methods that are child methods and map out the hierarchy. Also identify methods that interface with the rest of the code, these are used in the integration step.
4. Refactor the C code to avoid using dynamic memory. Some global variables may need adaptor code instead.
5. Rewrite the code in Whiley.
6. Refactor the Whiley code to ensure lists and objects, which are placed on the stack by the Whiley to C compiler, meet the scoping limits implicit to stack memory. Some may need to be global data and will need adaptor code.

5.1.1 Avoiding Dynamic Memory

Up to this point, the reasons for avoiding dynamic memory has been discussed, but not how it was avoided. Global values are used for a variety of reasons, for example the Crazyflie uses them to allow state to be shared between processes. There are two strategies used in

this project to avoid using dynamic memory in Whiley code; refactoring and using adaptor code.

Refactor
Global variables do not always need to be global. Where state is not shared between tasks, but is shared between a method and its child methods, the global variable can be refactored to be instantiated in the enclosing method. There were instances of this in the original Crazyflie stabilizer code. There were also instances of a global value being initialised and a pointer being passed to child methods for manipulation. In this case too it was possible to initialise the variable in a method and still pass pointers to child methods.

There is a memory trade-off. The effect on memory is to move storage from Heap memory to Stack memory (see Section 2.3.4), reducing the heap but increasing the size of the Stack Frame for each instance of the enclosing method on the stack. If only one instance of the methods stack frame will exist at any one time, there is no net memory cost. Otherwise a judgement call will be required, weighing up the cost on memory of having two or more instances Alternatively the instantiating method may be re-examined with a view to initialising the variable one level higher.

Adaptor Code
Globals variables cannot always be refactored. For example the Crazyflie stabilizer module uses a global boolean referenced by other Crazyflie tasks (it is true if the stabilizer task has successfully initialised, otherwise it is false). In this case adaptor code was used to facilitate a global variable. The adaptor code was written in C, taking advantage of the fact that C allows globals. Whiley's Foreign Function Interface facilitates using this code—which is discussed in the next section.

The adaptor code consists of a global variable declaration and getters and setters that can be used by Whiley as **native** methods. Listings 5.1 and 5.2 show a pattern that was used by Crazyflie tasks to initialise, in the process checking other tasks they depend on had already been initialised.

```
1  // native method declaration
2  native method isTest1() => bool
3  native method isTest2() => bool
4
5  method initTask() => bool:
6    bool result = isTest1() && isTest2()
7    if( !result ):
8      return false
9    // continue initializing
10   return true
```

Listing 5.1: Whiley calls a native method.

```
1  static bool test1 = false;
2
3  boolean isTest1(){
4    if( test1 ){ return true; }
5    test1 = doTheTestNow();
6    return test1;
7  }
```

Listing 5.2: The native adaptor code.

5.2 Integration

Integration is the process of enabling target code to communicate with host code. The need for this was identified in step 3 of the process outlined in Section 5.1 and adaptor code written in the host language may have been added in steps 4 and 6. Where target and host are written in two different languages—this is often facilitated by a Foreign Function Interface and adaptor code to provide datatype compatibility.

Foreign Function Interface
Foreign Function Interfaces (FFI) are a feature of many programming languages. Their purpose is to allow the program to invoke functions in other languages, often lower level languages to gain speed or other benefits. In this case the FFI allows the Whiley version of the stabilizer module to integrate with the remaining Crazyflie code in C.

The Whiley FFI consists of the keywords **native** and **export**. The **native** keyword allows Whiley code to describe a method signature that has been implemented in C code, while the **export** keyword allows Whiley to implement a method that can be used by the host application.

Listing 5.1 shows **native** method declarations in Whiley, that enables Whiley code to call the method written in C in Listing 5.2. Listing 5.3 shows an **export** method where the Whiley declaration includes the method body and expects the host application in Listing 5.4 to call it.

```
1 // export method declaration
2 export method stabilizerTest() => bool:
3   // do checks
4   return true
```

Listing 5.3: Whiley declares an export method.

```
1 // The Crazyflie commander module
2 boolean commanderInit(){
3   bool ok = stabilizerTest() && anotherTest();
4   if( !ok ) return false;
5   // continue initialising
6   return true;
7 }
```

Listing 5.4: And the C code uses the export method.

Datatype Compatibility
Integrating with existing C code can cause problems when an interface method requires a specific datatype, for example a uint_16. An unbounded Whiley **int** in this project translates to a bounded C **int**. But in several cases the interfacing method in C requires a uint_16. listing 5.5 shows the C method signature Whiley can support, while Listing 5.6 shows the method signature C is expecting.

```
1 // The method signature Whiley can support
2 int smallNumber( int x )
```

Listing 5.5: Whiley can support.

```
1 // The signature the C code complies with
2 uint_16 smallNumber( uint_16 x )
```

Listing 5.6: C expects.

This gap can be bridged using adaptor code and a simple pattern where Whiley calls a C method written for the signature in Listing 5.5, which translates the datatypes and calls the signature in Listing 5.6. This is illustrated in Listings 5.7 and 5.8 which show the Whiley **native** declaration for the method c_smallNumber()[1], and the implementation in C of the adaptor method c_smallNumber(), where the value is cast to a uint_16 and can now be used in the call to smallNumber(). The result is also cast back to an **int** for returning to Whiley.

[1]Note the prepended 'c_'. C does not support function overloading, meaning function names must be mangled, that is, made unique.

```
1  // Whiley adaptor method
2  // for smallNumber()
3  native c_smallNumber(int) => int
4
5  method main() => void:
6    int x = 5
7    int result = c_smallNumber(x)
8    ...
```

Listing 5.7: Native method calls an adapter written in C.

```
1  // C implementation of the adaptor method
2  static bool test1 = false;
3
4  int c_smallNumber(int x){
5    if( x < MIN_SHORT
6          || MAX_SHORT < x){
7      error(''Parameter value is out of range.'');
8    }
9    uint_16 y = (uint_16) x;
10   // use cast value in the integrated C method
11   y = smallNumber(y);
12   return (int) y;
13 }
```

Listing 5.8: The implementation of the adaptor code.

This chapter has outlined how to port an element of a host application to Whiley and then integrate the Whiley code with the remaining elements of the host application in C. Using the Whiley code created here, in the Whiley to C compiler outlined in Chapter 4, created the replacement stabilizer code which could now be used in a new binary file to be flashed to the Crazyflie. Ultimately this worked and the Crazyflie quad-copter was able to fly using Whiley code. The next chapter discusses the next stage, evaluating the performance of the new stabilizer code against the original.

Chapter 6

Evaluation

The motivation for this work, as outlined in Section 3.1, was to demonstrate using Whiley for programming embedded devices.

The project aim was to:

Demonstrate Whiley on an embedded system.

This was achieved by replacing existing C code on an embedded system with equivalent Whiley generated code and observing the embedded system functioning in a manner similar to the previous code.

Having succeeded in the core work, the question becomes:

Is Whiley code performance comparable to the original C code?

The purpose of this chapter is to detail efforts to investigate this.

6.1 The Tests

The experiments are designed to test performance factors related to the Crazyflie stabilizer algorithm. It may be recalled that the algorithm takes input from the pilot (the desired inputs) and the sensors (the actual inputs) (see Figure 2.3) and creates outputs for the four motors. The first experiment tests pilot inputs using the simple exercise of landing on a point. The second attempts to remove the pilot input and induce some flight instability for the algorithm to rectify. The last experiment measures algorithm speed. In each case the original Crazyflie code is used as a benchmark to compare the new code against.

6.1.1 Test Assumptions

Code Quality
The intention when replacing original C code with Whiley code, was to follow the structure of the original C code as closely as possible; this helped to mitigate any author bias or inclination to improve the code.

Memory Use
The use of static variables is eliminated in the replacement code, moving them in memory from the data segment to the stack. Some data structures have also been moved from heap memory, again to stack memory. These changes were not expected to make any appreciable influence on performance.

Figure 6.1: Crazyflie – at 4cm from the landing point.

6.1.2 Experiment 1

Pilot Landing Tests
Pilot inputs represent the desired inputs, while sensor inputs represent actual inputs. In normal flight both sets of inputs are used to ensure fast, predictable responses from the Crazyflie, that appear intuitive to the pilot. Any difference between the original and Whiley implementations will likely show up as decreased performance from the pilots perspective, resulting in increased variability when completing tasks.

The evaluation is a pilot controlled landing test. This provides the ability to take measurements from the point of aim, giving qualitative data to evaluate. The pilot task is to fly 2 metres away from the aim point and then return to land as close as possible to the point. A measurement is then taken from the aim point to the centre of the Crazyflie (see Figure **??**).

Confounding variables considered and mitigated are;

- **Air currents** An internal room was chosen with doors and windows closed.
- **Pilot distractions** All observers and other distractions were removed to provide a quiet environment
- **Pilot bias** The two binary files to be used were anonymised by a third party with no connection with the evaluation.
- **Improving pilot skill** The two anonymous binary files were flashed to the Crazyflie every five landings, to spread the effects of improving pilot skills over both sets of data.
- **Battery power** The Crazyflie has approximately 7 minutes of battery time. It was fully recharged after every five landings to keep the battery levels consistent.

The experiments generated two sets of 40 data points, measuring distance from the aim point in centimetres. The data provides the following metrics:

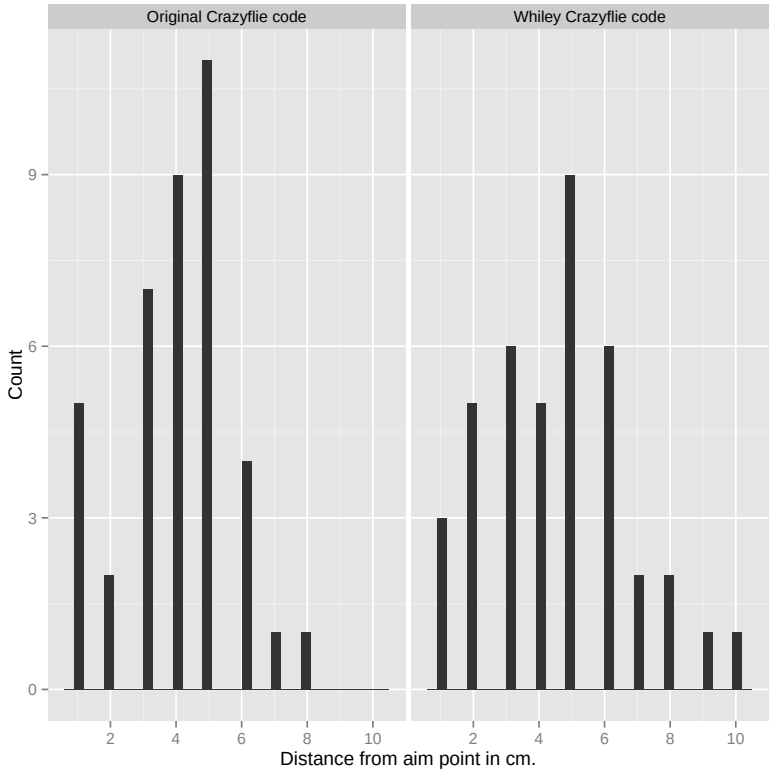


Figure 6.2: Histograms of 40 Pilot landing tests each.

	Original Crazyflie code	Whiley Crazyflie code
Mean	4.000	4.525
Median	4.000	5.000
Standard Deviation	1.69	2.18
High	8.00	10.00
Third quartile	5.00	6.00
First quartile	3.00	3.00
Low	1.00	1.00

As can be seen from the histograms in Figure 6.2, the results appear to be similar, with the Whiley code showing a little more variability.

6.1.3 Kolmogorov-Smirnov Test

The Kolmogorov-Smirnov test was chosen to determine whether the two data sets are statistically different [66]. The null hypothesis is that the two data sets are drawn from the same distribution. This hypothesis is rejected if the D value is greater than the value established by this formula at $\alpha = 0.05$:

$$\begin{aligned} d_{0.05} &= 1.36 * \sqrt{\frac{40+40}{40*40}} \\ &= 0.30 \end{aligned}$$

The result of the Kolmogorov-Smirnov test is D = 0.15 [1] which is less than $d_{0.05}$, meaning

[1] Established using the R statistical computing package.

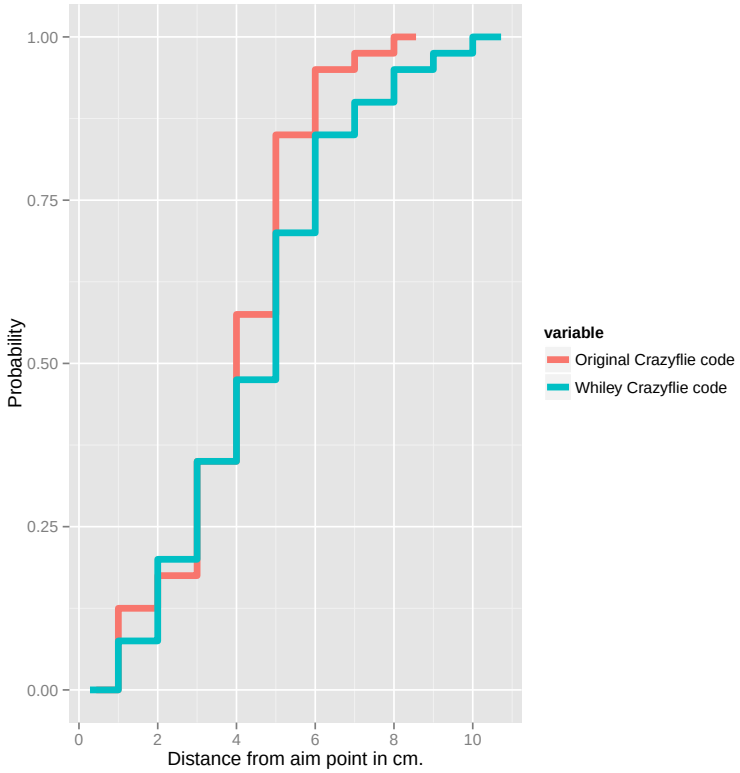


Figure 6.3: Aggregate landing results give the probability of landing within X cm.

the null hypothesis should not be rejected. The two distributions may be considered the same and the performance of the two implementations of the Crazyflie code are statistically similar at the 95% level of confidence.

Figure 6.3 plots the cumulative results against the probability along the Y axis. The D statistic represents the largest gap in the graph.

6.1.4 Experiment 2

Oscilation Tests

The sensors, when the Crazyflie is put under flight stress, provide inputs to the stabilizer algorithm enabling it to self level. This is an observable behaviour that occurs when for example, it collides with a ceiling, it passes through an air stream, it drops from a height and regains control or on a sudden changes of direction.

The evaluation intended to repeat the circumstances that create the oscillations, film the event and count the cycles until levelling had been achieved. This required a high speed camera and the ability to recreate the circumstances within the camera's field of view.

Technical difficulties have foiled all attempts to perform this test. Four cameras were tried, including two high speed cameras, one sourced from a lecturer (Chris Hollitt) , the other from the postgrad computer graphics department (Andrew Chalmers). It transpires that in order to get a sufficiently high frame rate on the most capable digital camera, the field of view is reduced to lighten the processors workload. For the better camera (the High Sensitivity USB 3.0 CMOS Camera from ThorLabs, set up in Figures 6.4a) and 6.4b, this resulted in a bounding box significantly smaller than expected, of 450mm x 600mm, 8.5 metres away from the camera (see Figure 6.4c).

(a) Thorlabs camera. **(b)** On tripod. **(c)** The field of view.

Figure 6.4: Thor High Sensitivity Camera—for high frame rates, needs a small fov.

(a) Crazyflie with two tethers. **(b)** Set up for tethered flight.

Figure 6.5: Crazyflie responded poorly, with or without the fan, when tethered.

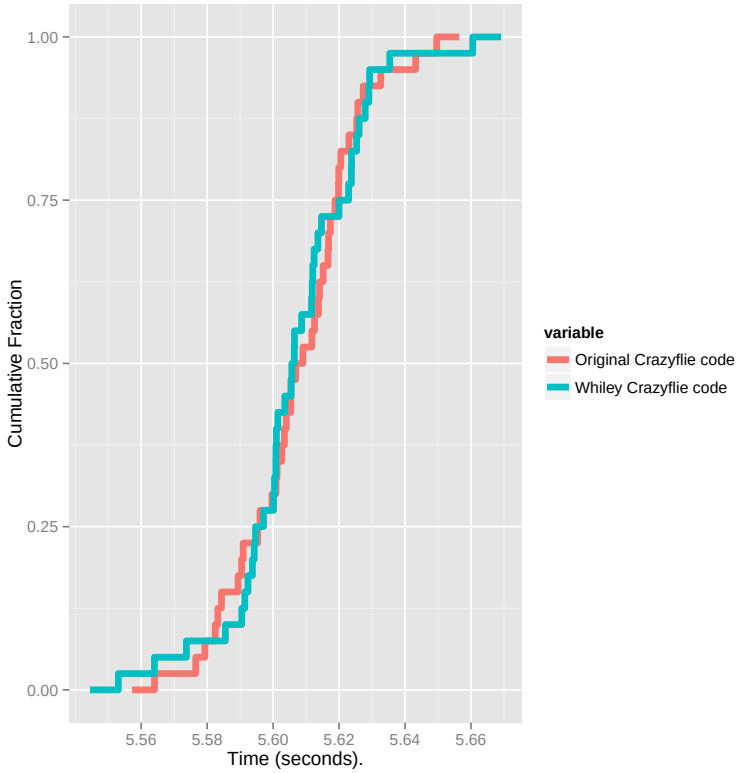


Figure 6.6: Runtime tests of Original code and Whiley code.

The problem then became one of getting observable activity within this bounding box. Dropping the Crazyflie from a height resulted in oscillations over a travel distance that exceeded the bounding box. Tests involving an airflow resulted in the Crazyflie being blown out of the bounding box, while tethering the Crazyflie by one, two and four points as shown in Figure 6.5a and 6.5b, resulted in abnormal behaviour. As a result of these problems, this experiment was abandoned.

6.1.5 Experiment 3

Software Performance Tests
This test is to determine any difference between the speed at which the original and new binary codes run.

To perform this test, a test harness was created to run the stabilizer algorithm on an x86_64 desktop computer running a Linux OS. Each version was cycled through its main algorithm 10,000,000 times per timed test and the test was taken 40 times each.

This test has at least two major limitations; it is not performed on the embedded device and the interface methods in both cases were only given stubs sufficient to allow the test to run. Consequently the tests may not reflect actual speeds on the device.

The experiments generated two sets of 40 data points, measuring the time taken in seconds. The data provides the following metrics:

The result of these tests was that the two binaries appear to be very similar in runtime as can be seen in Figure 6.6.

	Original Crazyflie code	Whiley Crazyflie code
Mean	5.604	5.607
Median	5.608	5.606
Standard Deviation	0.01811	0.01952
High	5.65	5.66
Third quartile	5.62	5.62
First quartile	5.60	5.60
Low	5.56	5.55

The Kolmogorov-Smirnov test was chosen to determine whether the two data sets are statistically different. The null hypothesis is that the two data sets are drawn from the same distribution. This hypothesis is rejected if the D value is greater than the value established by this formula at $\alpha = 0.05$:

$$\begin{aligned} d_{0.05} &= 1.36 * \sqrt{\frac{40+40}{40*40}} \\ &= 0.30 \end{aligned}$$

The result of the Kolmogorov-Smirnov test is D = 0.125 [2] which is less than $d_{0.05}$, meaning the null hypothesis should not be rejected. The two distributions may be considered the same and the performance of the two implementations of the Crazyflie code are statistically similar at the 95% level of confidence.

Based on these tests, it may be concluded that the two implementations are broadly similar in performance.

[2]Established using the R statistical computing package.

Chapter 7

Conclusions and Future Work

The goal was to demonstrate Whiley on an embedded system. Along the way it was anticipated the work would discover and highlight issues that face the Whiley project when being adapted for use in embedded programming. The project has succeeded in both endeavours. The Crazyflie flies on Whiley code and the Whiley projects body of knowledge in the embedded space has been expanded.

A feel for the complexity of the work might be gained from Appendix A which is a short version of the project's logbook. The most challenging aspect was finding how difficult it is to debug C on an embedded system. As a consequence, the practical part of the work over-ran by a significant margin.

A number of insights—some suspected, others new—have been demonstrated, which will no doubt prove useful in the future. For example one insight is the fragility of verified bytecode, its potential verified status was voided several times in this translation process. Another insight is that the strategies adopted by this compiler would add significantly to the cognitive workload of an embedded programmer. Overall however, most problems are not insurmountable and will likely be resolved by tackling the first three points listed here. The fourth point highlights a way to make the research a more productive:

- **Memory management.** Whiley assumes an operating system or algorithm that automatically frees up allocated heap memory. The largest friction point in this work was this assumption not translating to embedded systems. Future progress in this embedded system space may require a solution to this memory management problem.

- **Unbounded values.** Unbounded arithmetic values need to be bounded in the embedded space. The solution adopted in this work of merely translating to C ints and floats, would not be acceptable to the embedded community. A solution that may help, but was not implemented for this work, is to define a type in Whiley which is bound to a numerical range, as shown in List 7.1. This would be checked by the Whiley verifier and guarantee a safe cast to the equivalent C type.

- **Bytecode context.** The lack of context when translating bytecode resulted in some decisions having to be applied to the entire application. For example choosing between stack or heap memory for lists, requires context in order to avoid having to pick one approach for the entire application. One approach might be to add more context to bytecodes, other approaches may involve preprocessing the Whiley code to enable new strategies. For example; Escape Analysis (discussed in Section 3.2.2), pointer lifetimes[1] and pointer object models [67].

[1] Paul Koerbitz. Understanding Pointers, Ownership, and Lifetimes in Rust. Accessed Oct 2014. Retrieved from http://paulkoerbitz.de/posts/Understanding-Pointers-Ownership-and-Lifetimes-in-Rust.html

- **JTAG ICE Debugger.** Future efforts in this domain need to successfully tackle the JTAG debugger. It is an industry standard device and has the potential to save a lot of time by turning a multi-step and time consuming operation to move and run debug code, into operating in a debugging environment. The equipment is available at Victoria but not the experience in using it.

```
1  // Define a natural number
2  type nat is (int x) where x >= 0
3
4  // Define a C arithmetic type
5  type uint_16 is (int x)
6                where 0 <= x && x <= 65535
```

Listing 7.1: Defining custom Whiley types.

```
1  static bool test1 = false;
2
3  int c_smallNumber(int x){
4    if( x < MIN_SHORT
5          || MAX_SHORT < x){
6      error(''Parameter value is out of range.'');
7    }
8    short y = (short) x;
9    // use cast value in the integrated C method
10   y = smallNumber(y);
11   return (int) y;
12 }
```

Listing 7.2: Defining custom Whiley types.

The appendices hold three versions of the original Crazyflies stabilizer.c code; the first is the original (Appendix D) written in C, the second is the result of porting the original code to Whiley (Appendix E) and the last is the result of passing the Whiley bytecode through the Whiley to C compiler (Appendix F).

7.1 Future work

There are two major avenues for future work. One involves resolving the big problems mentioned already; develop or find a solution to memory management on memory constrained embedded devices; or resolve the unbounded to bounded values translation problem; or look at ways to include more context in bytecode or conversely examine other compiler strategies that can better extract the context from the source code.

The second major area of work could be repeating this exploration in another embedded device, perhaps in concert with resolving one of the first problems, for example:

- Create a new Crazyflie module using Whiley. Such as; live stream video from a camera attached to the Crazyflie or attach proximity sensors and develop an algorithm to avoid walls, floors and ceilings.
- Picking a device with a very simple RTOS and replacing the RTOS. This might be combined with developing a more refined Whiley to C compiler and interfacing with the hardware layer would provide a new challenge and may highlight new challenges.

7.2 Acknowledgements

I wish to thank Dr Pearce for the opportunity to work with him on this Whiley project and for the support, feedback and encouragement he has provided. I thank also my partner Megan, whom has been very supportive and extremely patient. Finally I wish to thank my colleagues; Sivan, Henry, Alex and Max, for keeping me grounded, providing encouragement and being sounding boards.

Appendix A

Logbook (Shortened)

This project involved an array of tasks in order to successfully achieve the outcomes. Many of which are related to the whole learning process. This is an overview for reference should a similar project be undertaken. The source is the project's daily journal.

1. **Quadcopter familiarisation** Get Quadcopter flying. Tues 11th March.
2. **RTOS** Research Real Time Operating Systems and FreeRTOS.
3. **Latex familiarization** start a Latex document for the proposal.
4. **Flash image** Download, compile, flash image to Quadcopter.
5. **Identify major components** FreeRTOS identified and researched.
6. **Makefile familiarisation** Makefiles researched, variations tried.
7. **GCC familiarisation** GCC researched.
8. **C language familiarisation** Hello World, followed by Conway's Game of Life exercise.
9. **Source code familiarization** Review source code, identify main components.
10. **Whiley familiarisation** Hello World followed by Conway's Game of Life exercise.
11. **Whiley to bytecode** Compiling to bytecode using WyC.
12. **Compilers familiarisation** Read Compilers course lecture slides.
13. **Device bricking** Directed to research and avoid locking up the embedded cpu.
14. **Identify Compiler Architecture** First draft.
15. **Whiley bytecode familiarization** Learning to identify state and behaviour.
16. **Compiler C Header and Library** Implemented.
17. **First compiled** Whiley to C program, Thursday 17th April.
18. **TDD** Created test harness and 51 sample Whiley programs.
19. **Any type** Created as a union of all primitive types using a union inside a struct.
20. **C pointer issues** referencing and dereferencing pointers and arrays.

21. **C method scope** Cannot pass arrays out of methods without using malloc.
22. **Malloc** Identified difficulties in how/when to free malloc with the compiler.
23. **Tuples and Records** Need malloc, resolved to ignore. (Solution later found for Records).
24. **First program run on Crazyflie** Flash LEDs and motor test, Thurs 8th May.
25. **Whiley Native and Extern** Used to marry Whiley with code base in other languages.
26. **Crazyflie code familiarisation** Read and develop overview.
27. **stabilizer.c** Chosen C file to replace, stripped it down and tested, rewritten in Whiley.
28. **cf_library.c** For global variables and interfacing Whiley with C.
29. **Crazyflie is dead** Battery was damaged and acid cooked the processor. Thur 4th July.
30. **Whiley test harness** Adapted, initial test run sees 19 of 610 pass.
31. **PID Controllers** Researched and pid.whiley created.
32. **Failed tests categorised** 145 passing, 465 failing for known reasons.
33. **Lambda functions** Required by native C app, implemented in compiler.
34. **Records** Records implemented.
35. **GCC Warnings** Discover -Wall -Wextra. Generates many warnings, resolved.
36. **Crazyflie is replaced** No significant project delays caused. Weds 23rd July.
37. **JTAG familiarisation** JTAG enables GDB debugging on embedded systems.
38. **GDB familiarisation** Tutorials.
39. **Flashing LEDs** Being used to debug, major memory bugs identified.
40. **Rewrite compiler** Have to reduce memory use, Removed 75 excess variable initialisations and removed a custom 41 byte compound type plus it's supporting code. Big job, Whiley test suite drops to 50 passing and takes a lot of work to rebuild.
41. **Whiley runtime assertions** Can be switched off, removes unneeded variables.
42. **Malloc is a huge problem** Start removing uses of malloc, much later figure out that the first use of malloc initialises a heap of 65kb.
43. **Whiley bug** Code `real a = 0`, initialises a as a **real** and then casts to an **int**. Subsequent uses cast it back to a **real**. Lots of unnecessary castes. Reported and fixed.
44. **C Strings** Research into different implementations, malloc and char[ ] both cause problems that literals do not seem to have.
45. **Whiley New** Whiley places **new** objects on the heap, instead adopt the use of stack memory to avoid malloc issues (Last use of malloc removed).
46. **It works!** Flying the Crazyflie with the new stabilizer.c code. Mon 1st Sept.
47. **Port 2 more files** Port controller.c and pid.c to Whiley code, debug and integrate with original Crazyflie code.
48. **Job done** Flying the Crazyflie with the Whiley stabilizer algorithm. Sat 6th Sept.

Appendix B

Whiley bytecode

Whiley is similar to the Java programing language in that it compiles to bytecode. Bytecode is optimised by the Whiley WyC compiler as it is compiled, making it smaller, faster and more efficient. In addition Whiley verified code generates Whiley verified bytecode. There are a wide variety of bytecode types, for example the Assign bytecode provides instructions to assign a value held by variable A, to variable B.

This project is interested in Whiley bytecode only if they directly assist the project aim of demonstrating Whiley on an embedded system. This means there are numerous bytecodes that have not been implemented in the projects compiler. What follows is a list of bytecodes, with a brief explanation of what the bytecode does from the Whiley API. Bytecodes implemented by the project are marked with an asterix. Most show actual bytecode examples from this project.

1. **.label21** * Label bytecode, destination for a goto
2. **add %29 = %0, %1 : int** * A binary operation which reads two numeric values from the operand registers, performs an operation on them and writes the result to the target register.
3. **append %14 = %1, %13 : [int]** Append one list to another.
4. **assertge %0, %2 "constraint not satisfied" : int** Reads two operand registers, compares their values and raises an assertion failure with the given message if comparison is false.
5. **AssertOrAssume** An abstract class representing either an assert or assume bytecode.
6. **assign %39 = %25 : int** * Copy the contents from a given operand register into a given target register.
7. **assumege %1, %8 "message" : int** Reads two operand registers, compares their values and raises an assertion failure with the given message, if comparison is false.
8. **BinListOp** Reads the (effective) list values from two operand registers, performs an operation.
9. **BinSetOp** A binary operation which reads two set values from the operand registers, performs an operation on them and writes the result to the target register.
10. **const %10 = 0 : int** * Writes a constant value to a target register.

11. **convert %29 = %29 any : string** * Reads a value from the operand register, converts it to a given type and writes the result to the target register.

12. **Debug** Read a string from the operand register and prints it to the debug console.

13. **deref %261 = %65 : &int** * Reads a reference value from the operand register and dereferences it.

14. **div %26 = %24, %25 : real** * A binary operation which reads two numeric values from the operand registers, performs an operation on them and writes the result to the target register.

15. **end label18** * Marks the end of a loop block.

16. **fieldload %13 = %12 kd :** {**real integ,real kd,real ki**} * Reads a record value from an operand register, extracts the value of a given field and writes this to the target register.

17. **forall %7 in %5 () : [int]** * Pops a set, list or map from the stack and iterates over every element it contains.

18. **goto label21** * Branches unconditionally to the given label.

19. **ifeq %18, %19 goto label20 : bool** * Branches conditionally to the given label based on the result of a runtime type test against a value from the operand register.

20. **ifge %13, %22 goto label4 : int** * Branches conditionally to the given label based on the result of a runtime type test against a value from the operand register.

21. **ifle %0, %4 goto label24 : int** * Branches conditionally to the given label by reading the values from two operand registers and comparing them.

22. **IfIs** Branches conditionally to the given label based on the result of a runtime type test against a value from the operand register.

23. **iflt %19, %21 goto label22 : int** * Branches conditionally to the given label based on the result of a runtime type test against a value from the operand register.

24. **ifne %0, %6 goto label1 : int** * Branches conditionally to the given label based on the result of a runtime type test against a value from the operand register.

25. **indexof %61 = %0, %60 : [real]** * Reads an effective list or map from the source (left) operand register, and a key value from the key (right) operand register and returns the value associated with that key.

26. **indirectinvoke %10(%11) : method(any) =>void** * Represents an indirect function call.

27. **Invert** Corresponds to a bitwise inversion operation, which reads a byte value from the operand register, inverts it and writes the result to the target resgister.

28. **invoke %(%38, %39) stabilizer:cf_motorsSetRatio : method(int,int) =>void** * Corresponds to a function or method call whose parameters are read from zero or more operand registers.

29. **Label** * Represents the labelled destination of a branch or loop statement.

30. **lambda %2 = () stabilizer:stabilizerTask : method() =>void** * Represents a pointer to a method.
31. **lengthof %5 = %0 : [byte]** * Reads a collections length, assigns it to a target register.
32. **ListLVal** An LVal with list type.
33. **loop (%4)** * Represents a block of code which loops continuously until the condition is met.
34. **LVal<T>** Represents a type which may appear on the left of an assignment expression.
35. **MapLVal** An LVal with map type.
36. **Move** Moves the contents of a given operand register into a given target register.
37. **mul %71 = %67, %70 : real** * A binary operation which reads two numeric values from the operand registers, performs an operation on them and writes the result to the target register.
38. **neg %262 = %261 : int** * Create negative number
39. **newlist %16 = (%13, %14, %15) : [real]** * Constructs a new list value from the values given by zero or more operand registers.
40. **NewMap** Constructs a map value from zero or more key-value pairs on the stack.
41. **newobject %65 = %64 : &int** * Instantiate a new object from the value in a given operand register, and write the result (a reference to that object) to a given target register.
42. **newrecord %169 = (%155, %156, %157) :** {**real integ,real kd,real ki**} * Constructs a new record value from the values of zero or more operand register, each of which is associated with a field name.
43. **newset %19 = (%14, %15, %16) : int** Constructs a new set value from the values given by zero or more operand registers.
44. **NewTuple** Constructs a new tuple value from the values given by zero or more operand registers.
45. **nop** * Represents a no-operation bytecode which, as the name suggests, does nothing.
46. **Not** Read a boolean value from the operand register, inverts it and writes the result to the target register.
47. **range %6 = %3, %5 : [int]** * Range of values from one parameter to the next.
48. **RecordLVal** An LVal with record type.
49. **ReferenceLVal** An LVal with list type.
50. **return %1 : int** * Returns from the enclosing function or method, possibly returning a value.
51. **sappend %26 = %24, %1 : string** A binary operation which reads two string values from the operand registers, performs an operation (append) on them and writes the result to the target register.

52. **StringLVal** An LVal with string type.

53. **sub %31 = %29, %3 : int** * A binary operation which reads two numeric values from the operand registers, performs an operation on them and writes the result to the target register.

54. **SubList** Reads the (effective) list value from a source operand register, and the integer values from two index operand registers, computes the sublist and writes the result back to a target register.

55. **SubString** Reads the string value from a source operand register, and the integer values from two index operand registers, computes the substring and writes the result back to a target register.

56. **switch %0 1->label1, -1->label2, *->label0** * Performs a multi-way branch based on the value contained in the operand register.

57. **throw %4 : string** Throws an exception containing the value in the given operand register.

58. **trycatch string->label1** Represents a try-catch block within which specified exceptions will caught and processed within a handler.

59. **TryEnd** Marks the end of a try-catch block.

60. **TupleLoad** Read a tuple value from the operand register, extract the value it contains at a given index and write that to the target register.

61. **UnArithOp** Read a number (int or real) from the operand register, perform a unary arithmetic operation on it.

62. **update (*%0).dt %2 : &{real deriv,real desired,real dt} ->&{real deriv,real desired,real dt}** Pops a compound structure, zero or more indices and a value from the stack and updates the compound structure with the given value.

63. **Void** The void bytecode is used to indicate that the given register(s) are no longer live.

Appendix C

The Whiley Test Suite

The Whiley test suite consists of a test harness and 610 Whiley scripts with matching oracle answers.

When working with the test suite, as each test was examined for insights on why it failed, it was labelled with a single failure cause. This may not be the only cause, just the one perceived to be most problematic at the time. As such, the following statistics should be considered indicative only.

Total Tests	610
Passing tests	114
Records	69
Unions	78
Try-catch	13
Array size	119
Tuples	14
Sets	72
Big number	5
Not WyC compiled	23
Bytes	11
Range	8
Constants	5
Arrays	17
Dictionary	17
Correctly print real (eg: 12/23)	10
Miscellaneous	35

Appendix D

Original stabilizer.c Code

The stabilizer.c code is available on Github [1]. The version presented here is the result of analysing and stripping the code of its hover functionality. This formed the working version for this project.

The other files used were controller.c and pid.c. These are also available for viewing on Github [2].

```
1  /*
2   * Crazyflie Firmware
3   *
4   * Copyright (C) 2011−2012 Bitcraze AB
5   *
6   * This program is free software: you can redistribute it and/or modify
7   * it under the terms of the GNU General Public License as published by
8   * the Free Software Foundation, in version 3.
9   *
10  * This program is distributed in the hope that it will be useful,
11  * but WITHOUT ANY WARRANTY; without even the implied warranty of
12  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
13  * GNU General Public License for more details.
14  *
15  * You should have received a copy of the GNU General Public License
16  * along with this program. If not, see <http://www.gnu.org/licenses/>.
17  */
18
19 #include "stm32f10x_conf.h"
20 #include "FreeRTOS.h"
21 #include "task.h"
22
23 #include "system.h"
24 #include "stabilizer.h"
25 #include "commander.h"
26 #include "controller.h"
27 #include "sensfusion6.h"
28 #include "imu.h"
29 #include "motors.h"
30 #include "log.h"
31
32 uint32_t motorPowerM4;
33 uint32_t motorPowerM2;
```

[1] Accessed Sept 2014. https://github.com/bitcraze/crazyflie-firmware/blob/master/modules/src/stabilizer.c

[2] Accessed Sept 2014. https://github.com/bitcraze/crazyflie-firmware/tree/master/modules/src

```
34 uint32_t motorPowerM1;
35 uint32_t motorPowerM3;
36
37 static bool isInit;
38
39 static void distributePower(const uint16_t thrust, const int16_t roll, const int16_t pitch, const int16_t yaw);
40 static uint16_t limitThrust(int32_t value);
41 static void stabilizerTask(void* param);
42
43 void stabilizerInit(void)
44 {
45   if(isInit)
46     return;
47
48   motorsInit();
49   imu6Init();
50   sensfusion6Init();
51   controllerInit();
52
53   xTaskCreate(stabilizerTask, (const signed char * const)"STABILIZER",
54               /*2*configMINIMAL_STACK_SIZE*/200, NULL, /*Piority*/2, NULL);
55
56   isInit = TRUE;
57 }
58
59 bool stabilizerTest(void)
60 {
61   bool pass = true;
62
63   pass &= motorsTest();
64   pass &= imu6Test();
65   pass &= sensfusion6Test();
66   pass &= controllerTest();
67
68   return pass;
69 }
70
71 static void stabilizerTask(void* param)
72 {
73   static Axis3f gyro; // Gyro axis data in deg/s
74   static Axis3f acc; // Accelerometer axis data in mG
75   static Axis3f mag; // Magnetometer axis data in testla
76
77   static float eulerRollActual;
78   static float eulerPitchActual;
79   static float eulerYawActual;
80   static float eulerRollDesired;
81   static float eulerPitchDesired;
82   static float eulerYawDesired;
83   static float rollRateDesired = 0;
84   static float pitchRateDesired = 0;
85   static float yawRateDesired = 0;
86
87   RPYType rollType;
88   RPYType pitchType;
89   RPYType yawType;
```

```
90
91    uint16_t actuatorThrust;
92    int16_t actuatorRoll;
93    int16_t actuatorPitch;
94    int16_t actuatorYaw;
95
96    uint32_t attitudeCounter = 0;
97    uint32_t lastWakeTime;
98
99    vTaskSetApplicationTaskTag(0, (void*)/*TASK_STABILIZER_ID_NBR*/3);
100
101   //Wait for the system to be fully started to start stabilization loop
102   systemWaitStart();
103
104   lastWakeTime = xTaskGetTickCount ();
105
106   while(1)
107   {
108     vTaskDelayUntil(&lastWakeTime, (unsigned int)
109       ((/*configTICK_RATE_HZ*/ /*( portTickType ) cast to short*/ 1000 / /*IMU_UPDATE_FREQ*/500)) ); // 500Hz
110
111     // Magnetometer not yet used more then for logging.
112     imu9Read(&gyro, &acc, &mag);
113
114     if (imu6IsCalibrated())
115     {
116       commanderGetRPY(&eulerRollDesired, &eulerPitchDesired, &eulerYawDesired);
117       commanderGetRPYType(&rollType, &pitchType, &yawType);
118
119       // 250HZ
120       if (++attitudeCounter >= /*ATTITUDE_UPDATE_RATE_DIVIDER*/2)
121       {
122         sensfusion6UpdateQ(gyro.x, gyro.y, gyro.z, acc.x, acc.y, acc.z, /*FUSION_UPDATE_DT*/(float)
123           (1.0/(/*IMU_UPDATE_FREQ*/500 / /*ATTITUDE_UPDATE_RATE_DIVIDER*/2)) );
124         sensfusion6GetEulerRPY(&eulerRollActual, &eulerPitchActual, &eulerYawActual);
125
126         controllerCorrectAttitudePID(eulerRollActual, eulerPitchActual, eulerYawActual,
127                                      eulerRollDesired, eulerPitchDesired, -eulerYawDesired,
128                                      &rollRateDesired, &pitchRateDesired, &yawRateDesired);
129         attitudeCounter = 0;
130       }
131
132       if (rollType == RATE) { rollRateDesired = eulerRollDesired; }
133       if (pitchType == RATE) { pitchRateDesired = eulerPitchDesired; }
134       if (yawType == RATE) { yawRateDesired = -eulerYawDesired; }
135
136       controllerCorrectRatePID(gyro.x, -gyro.y, gyro.z, rollRateDesired, pitchRateDesired, yawRateDesired);
137       controllerGetActuatorOutput(&actuatorRoll, &actuatorPitch, &actuatorYaw);
138       commanderGetThrust(&actuatorThrust);
139
140       if (actuatorThrust > 0)
141       {
142         distributePower(actuatorThrust, actuatorRoll, actuatorPitch, -actuatorYaw);
143       } else {
144         distributePower(0, 0, 0, 0);
145         controllerResetAllPID();
```

```
146         }
147       }
148     }
149 }
150
151 static void distributePower(const uint16_t thrust, const int16_t roll, const int16_t pitch, const int16_t yaw)
152 {
153  // QUAD_FORMATION_NORMAL
154   motorPowerM1 = limitThrust(thrust + pitch + yaw);
155   motorPowerM2 = limitThrust(thrust - roll - yaw);
156   motorPowerM3 = limitThrust(thrust - pitch + yaw);
157   motorPowerM4 = limitThrust(thrust + roll - yaw);
158
159   motorsSetRatio(/*MOTOR_M1*/0, motorPowerM1);
160   motorsSetRatio(/*MOTOR_M2*/1, motorPowerM2);
161   motorsSetRatio(/*MOTOR_M3*/2, motorPowerM3);
162   motorsSetRatio(/*MOTOR_M4*/3, motorPowerM4);
163 }
164
165 static uint16_t limitThrust(int32_t value)
166 {
167   if(value > /*UINT16_MAX*/65535)
168   {
169     value = /*UINT16_MAX*/65535;
170   } else if(value < 0) {
171     value = 0;
172   }
173   return (uint16_t)value;
174 }
```

Appendix E

Whiley version of stabilizer.c

The stabilizer.whiley code manually ported from stabilizer.c.

```
1  import whiley.lang.System
2  import * from controller
3
4  // pid model -- a record
5  public type PidObject is {
6      real desired,
7      real error,
8      real prevError,
9      real integ,
10     real deriv,
11     real kp,
12     real ki,
13     real kd,
14     real outP,
15     real outI,
16     real outD,
17     real iLimit,
18     real iLimitLow,
19     real dt
20     }
21
22 //== Tests ==
23 native method motorsTest() => bool
24 native method imu6Test() => bool
25 native method sensfusion6Test() => bool
26 //native method controllerTest() => bool
27
28 //== Initialize ==
29 native method motorsInit()
30 native method imu6Init()
31 native method sensfusion6Init()
32 //native method controllerInit([&PidObject] pidArray)
33 native method isStabilizerInit() => bool
34
35 //== Simple methods, no parameters ==
36 native method systemWaitStart()
37 native method cf_lib_xTaskGetTickCount() => int
38 native method imu6IsCalibrated() => bool
39 //native method controllerResetAllPID()
40
```

```
41 native method cf_lib_LHS_Equals_Neg_RHS( &real yawRateDesired, &real eulerYawDesired)
42
43 //==============
44 //== FreeRTOS ==
45   // portBASE_TYPE xTaskCreate( pdTASK_CODE pvTaskCode, const char * const pcName, unsigned short usStackDepth, v
46 native method cf_lib_xTaskCreate(method() => void stabilizerTask, string b, int c, int d, int e, int f) => void
47
48   // void vTaskSetApplicationTaskTag( xTaskHandle xTask, pdTASK_HOOK_CODE pxHookFunction ) PRIVILEGED_FUNCTI
49   // typedef void * xTaskHandle;
50   // pdTASK_HOOK_CODE is used as a void*, replace with void* ?
51 native method cf_lib_vTaskSetApplicationTaskTag(int p, int taskStabilizerIdNmr)
52
53   //void vTaskDelayUntil( portTickType * const pxPreviousWakeTime, portTickType xTimeIncrement ) PRIVILEGED_FUNCT
54 native method cf_lib_vTaskDelayUntil( int lastWakeTime, int xTimeIncrement ) => int
55
56 //================
57 //== i/o operations ==
58   //void imu9Read(Axis3f* gyroOut, Axis3f* accOut, Axis3f* magOut);
59 native method cf_lib_imu9Read( &[real] gyro, &[real]_acc, &[real] mag) //done
60
61   //void sensfusion6UpdateQ(float gx, float gy, float gz, float ax, float ay, float az, float dt);
62 native method cf_lib_sensfusion6UpdateQ( &[real] gyro, &[real] acc, real dt)
63
64   //void sensfusion6GetEulerRPY(float* roll, float* pitch, float* yaw);
65 native method sensfusion6GetEulerRPY( &real eulerRollActual, &real eulerPitchActual, &real eulerYawActual) //done
66
67 //== commander.c ==
68   //void commanderGetThrust(uint16_t* thrust);
69 native method cf_lib_commanderGetThrust(&int actuatorThrust) => void // done
70   //void commanderGetRPY(float* eulerRollDesired, float* eulerPitchDesired, float* eulerYawDesired);
71 native method commanderGetRPY(&real eulerRollDesired, &real eulerPitchDesired, &real eulerYawDesired)
72
73   //void commanderGetRPYType(RPYType* rollType, RPYType* pitchType, RPYType* yawType);
74 native method cf_lib_commanderGetRPYType(&string rollType, &string pitchType, &string yawType)
75
76 native method cf_lib_motorsSetRatio(int motor, int power)
77
78 //==========
79 export method stabilizerTest() => bool:
80   bool pass = true
81
82   pass = pass && motorsTest()
83   pass = pass && imu6Test()
84   pass = pass && sensfusion6Test()
85   pass = pass && controllerTest()
86   return pass
87
88 //==========
89 export method stabilizerInit() => void:
90   if(isStabilizerInit()):
91     return
92
93   motorsInit()
94   imu6Init()
95   sensfusion6Init()
96   // controllerInit() // moved to after pid initialisations below ~ln 130
```

```
97
98    // create the stabilizer task. Places the task into the FreeRTOS task que/s.
99    cf_lib_xTaskCreate(&stabilizerTask, /*(const signed char * const)*/ "STABILIZER", 200, /*null*/0, /*Piority*/2, /*null*/
100
101 //=====
102 // This sets up and contains the loop that the stabilzer task runs
103 method stabilizerTask() => void:
104    //========== INITIALISE ============
105    &[real] gyro = new [0.0, 0.0, 0.0]
106    &[real] acc = new [0.0, 0.0, 0.0]
107    &[real] mag = new [0.0, 0.0, 0.0]
108
109    &real eulerRollActual = new 0.0
110    &real eulerPitchActual = new 0.0
111    &real eulerYawActual = new 0.0
112    &real eulerRollDesired = new 0.0
113    &real eulerPitchDesired = new 0.0
114    &real eulerYawDesired = new 0.0
115    &real rollRateDesired = new 0.0
116    &real pitchRateDesired = new 0.0
117    &real yawRateDesired = new 0.0
118
119    &string rollType = new "ANGLE"
120    &string pitchType = new "ANGLE"
121    &string yawType = new "ANGLE"
122
123    &int actuatorThrust = new 0 // was uint16
124    &int actuatorRoll = new 0 // was int16
125    &int actuatorPitch = new 0
126    &int actuatorYaw = new 0
127
128    int attitudeCounter = 0 // was uint32_t
129    int lastWakeTime // was uint32_t
130
131    //============================
132    // Refactored controller code.
133    // Object declarations and controllerInit() inserted here to use stack declarations
134    // and avoid issues with globals and heap declarations.
135
136    &PidObject pidRollRate = new { desired: 0.0, error: 0.0, prevError: 0.0, integ: 0.0, deriv: 0.0, kp: 0.0, ki: 0.0, kd: 0.0, outP
137    &PidObject pidPitchRate = new { desired: 0.0, error: 0.0, prevError: 0.0, integ: 0.0, deriv: 0.0, kp: 0.0, ki: 0.0, kd: 0.0, ou
138    &PidObject pidYawRate = new { desired: 0.0, error: 0.0, prevError: 0.0, integ: 0.0, deriv: 0.0, kp: 0.0, ki: 0.0, kd: 0.0, outP
139    &PidObject pidRoll = new { desired: 0.0, error: 0.0, prevError: 0.0, integ: 0.0, deriv: 0.0, kp: 0.0, ki: 0.0, kd: 0.0, outP: 0.0
140    &PidObject pidPitch = new { desired: 0.0, error: 0.0, prevError: 0.0, integ: 0.0, deriv: 0.0, kp: 0.0, ki: 0.0, kd: 0.0, outP: 0
141    &PidObject pidYaw = new { desired: 0.0, error: 0.0, prevError: 0.0, integ: 0.0, deriv: 0.0, kp: 0.0, ki: 0.0, kd: 0.0, outP: 0.0
142
143    [&PidObject] pidArray = [pidRollRate, pidPitchRate, pidYawRate, pidRoll, pidPitch, pidYaw]
144
145    controllerInit(pidArray)
146
147    //== end refactored controller code ==
148
149    cf_lib_vTaskSetApplicationTaskTag(0, /*TASK_STABILIZER_ID_NBR*/3) // FreeRTOSConfig.h #define TASK_STABILIZ
   3
150
151    systemWaitStart()
```

```
152
153     lastWakeTime = cf_lib_xTaskGetTickCount()
154
155     //========== START LOOP =============
156     while(true):
157       //vTaskDelayUntil(&lastWakeTime, (unsigned int)((/*configTICK_RATE_HZ*/ /*( portTickType ) cast to short*/ 1000 /
158       lastWakeTime = cf_lib_vTaskDelayUntil(lastWakeTime, 2)
159
160       cf_lib_imu9Read(gyro, acc, mag)
161
162       if(imu6IsCalibrated()):
163         commanderGetRPY(eulerRollDesired, eulerPitchDesired, eulerYawDesired)
164
165         cf_lib_commanderGetRPYType(rollType, pitchType, yawType)
166
167         attitudeCounter = attitudeCounter + 1
168         if(attitudeCounter >= 2):
169           real fusion_update_dt = 1.0/(500.0 / 2.0)
170           cf_lib_sensfusion6UpdateQ( gyro, acc, fusion_update_dt)
171           sensfusion6GetEulerRPY(eulerRollActual, eulerPitchActual, eulerYawActual)
172
173           controllerCorrectAttitudePID(
174                   *eulerRollActual, *eulerPitchActual, *eulerYawActual,
175                   *eulerRollDesired, *eulerPitchDesired, -(*eulerYawDesired),
176                   rollRateDesired, pitchRateDesired, yawRateDesired,
177                   pidRoll, pidPitch, pidYaw)
178
179           attitudeCounter = 0
180
181         // dropped several redundent if statements
182         cf_lib_LHS_Equals_Neg_RHS( yawRateDesired, eulerYawDesired )
183
184         controllerCorrectRatePID(
185             *gyro,
186             *rollRateDesired, *pitchRateDesired, *yawRateDesired,
187             actuatorRoll, actuatorPitch, actuatorYaw,
188             pidRollRate, pidPitchRate, pidYawRate
189             )
190
191         cf_lib_commanderGetThrust(actuatorThrust)
192
193         if((*actuatorThrust) > 0):
194           distributePower(*actuatorThrust, *actuatorRoll, *actuatorPitch, -(*actuatorYaw))
195         else:
196           distributePower(0, 0, 0, 0)
197           controllerResetAllPID(pidArray)
198
199     //========== END LOOP =============
200
201   //=====
202   method distributePower(int thrust, int roll, int pitch, int yaw): // takes uint16
203     int motorPowerM1 = limitThrust(thrust + pitch + yaw)
204     int motorPowerM2 = limitThrust(thrust - roll - yaw)
205     int motorPowerM3 = limitThrust(thrust - pitch + yaw)
206     int motorPowerM4 = limitThrust(thrust + roll - yaw)
207
```

```
208     cf_lib_motorsSetRatio(/*MOTOR_M1*/0, motorPowerM1)
209     cf_lib_motorsSetRatio(/*MOTOR_M2*/1, motorPowerM2)
210     cf_lib_motorsSetRatio(/*MOTOR_M3*/2, motorPowerM3)
211     cf_lib_motorsSetRatio(/*MOTOR_M4*/3, motorPowerM4)
212
213   //=====
214   method limitThrust(int v) => int: // converts an uint32 to a uint16
215     int value = v
216     int uint16_Max = 65535
217     if(value > uint16_Max):
218       value = uint16_Max
219     else if(value < 0):
220       value = 0
221     return value
```

Appendix F

Whiley generated C code for stabilizer.c

This is the output from the Whiley to C compiler, the input was stabilizer.wyil—the Whiley bytecode for stabilizer.whiley.

```
1  #define LIBRARY_TESTING false
2
3  #include <stdio.h>
4  #include <stdbool.h>
5
6  #define STRINGMAX 10 // used in snprint functions
7  #define real float // can be changed to suit application
8  #include "stm32f10x_conf.h"
9  #include <math.h>
10 #include "FreeRTOS.h"
11 #include "task.h"
12 #include "led.h"
13 #include "motors.h"
14 #include "task.h"
15 #include "system.h"
16 #include "stabilizer.h"
17 #include "commander.h"
18 #include "sensfusion6.h"
19 #include "param.h"
20 #include "imu.h"
21 #include "log.h"
22 #include "_whiley/mattCompiler.h"
23 #include "_whiley/mattCompiler_library.c"
24 #include "_whiley/cf_Lib.c"
25
26 typedef struct {
27     real deriv;
28     real desired;
29     real dt;
30     real error;
31     real iLimit;
32     real iLimitLow;
33     real integ;
34     real kd;
35     real ki;
36     real kp;
37     real outD;
```

```
38        real outI;
39        real outP;
40        real prevError;
41        } PidObject;
42
43  bool stabilizerTest ( void );
44  void stabilizerInit ( void );
45  void stabilizerTask ( void );
46  void distributePower ( int, int, int, int );
47  int limitThrust ( int );
48
49  void controllerInit ( PidObject** );
50  bool controllerTest ( void );
51  void controllerCorrectRatePID ( real*, real, real, real, int*, int*, int*, PidObject*, PidObject*, PidObject* );
52  void controllerCorrectAttitudePID ( real, real, real, real, real, real, real*, real*, real*, PidObject*, PidObject*, PidObject* );
53  void controllerResetAllPID ( PidObject** );
54
55
56  bool stabilizerTest (void){
57    bool a1 = true;
58    bool a3 = true;
59    if ( a1 == a3 ) { goto label0; };
60    goto label1;
61    label0: ;
62    bool a4 = motorsTest ( );
63    bool a5 = true;
64    if ( a4 == a5 ) { goto label2; };
65    label1: ;
66    bool a6 = false;
67    goto label3;
68    label2: ;
69    a6 = true;
70    label3: ;
71    bool a8 = true;
72    if ( a6 == a8 ) { goto label4; };
73    goto label5;
74    label4: ;
75    bool a9 = imu6Test ( );
76    bool a10 = true;
77    if ( a9 == a10 ) { goto label6; };
78    label5: ;
79    bool a11 = false;
80    goto label7;
81    label6: ;
82    a11 = true;
83    label7: ;
84    bool a13 = true;
85    if ( a11 == a13 ) { goto label8; };
86    goto label9;
87    label8: ;
88    bool a14 = sensfusion6Test ( );
89    bool a15 = true;
90    if ( a14 == a15 ) { goto label10; };
91    label9: ;
92    bool a16 = false;
93    goto label11;
```

```
94    label10: ;
95    a16 = true;
96    label11: ;
97    bool a18 = true;
98    if ( a16 == a18 ) { goto label12; };
99    goto label13;
100   label12: ;
101   bool a19 = controllerTest ( );
102   bool a20 = true;
103   if ( a19 == a20 ) { goto label14; };
104   label13: ;
105   bool a21 = false;
106   goto label15;
107   label14: ;
108   a21 = true;
109   label15: ;
110   return a21;
111 }
112
113 void stabilizerInit (void){
114   bool a0 = isStabilizerInit ( );
115   bool a1 = true;
116   if ( a0 == a1 ) { goto label16; };
117   goto label17;
118   label16: ;
119   return;
120   label17: ;
121   motorsInit ( );
122   imu6Init ( );
123   sensfusion6Init ( );
124   void (*a2)() = &stabilizerTask;
125   char * a3 = "STABILIZER";
126   int a4 = 200;
127   int a5 = 0;
128   int a6 = 2;
129   int a7 = 0;
130   cf_lib_xTaskCreate ( a2, a3, a4, a5, a6, a7 );
131   return;
132 }
133
134 void stabilizerTask (void){
135   real a1 = 0.0;
136   real a2 = 0.0;
137   real a3 = 0.0;
138   real a4[3];
139   a4[0] = a1;
140   a4[1] = a2;
141   a4[2] = a3;
142   real *a5 = &(a4[0]);
143   real a7 = 0.0;
144   real a8 = 0.0;
145   real a9 = 0.0;
146   real a10[3];
147   a10[0] = a7;
148   a10[1] = a8;
149   a10[2] = a9;
```

```
150     real *a11 = &(a10[0]);
151     real a13 = 0.0;
152     real a14 = 0.0;
153     real a15 = 0.0;
154     real a16[3];
155     a16[0] = a13;
156     a16[1] = a14;
157     a16[2] = a15;
158     real *a17 = &(a16[0]);
159     real a19 = 0.0;
160     real *a20 = &a19;
161     real a22 = 0.0;
162     real *a23 = &a22;
163     real a25 = 0.0;
164     real *a26 = &a25;
165     real a28 = 0.0;
166     real *a29 = &a28;
167     real a31 = 0.0;
168     real *a32 = &a31;
169     real a34 = 0.0;
170     real *a35 = &a34;
171     real a37 = 0.0;
172     real *a38 = &a37;
173     real a40 = 0.0;
174     real *a41 = &a40;
175     real a43 = 0.0;
176     real *a44 = &a43;
177     char * a46 = "ANGLE";
178     char *a47 = a46;
179     char * a49 = "ANGLE";
180     char *a50 = a49;
181     char * a52 = "ANGLE";
182     char *a53 = a52;
183     int a55 = 0;
184     int *a56 = &a55;
185     int a58 = 0;
186     int *a59 = &a58;
187     int a61 = 0;
188     int *a62 = &a61;
189     int a64 = 0;
190     int *a65 = &a64;
191     int a67 = 0;
192     real a70 = 0.0;
193     real a71 = 0.0;
194     real a72 = 0.0;
195     real a73 = 0.0;
196     real a74 = 0.0;
197     real a75 = 0.0;
198     real a76 = 0.0;
199     real a77 = 0.0;
200     real a78 = 0.0;
201     real a79 = 0.0;
202     real a80 = 0.0;
203     real a81 = 0.0;
204     real a82 = 0.0;
205     real a83 = 0.0;
```

```
206 PidObject a84 = { a70, a71, a72, a73, a74, a75, a76, a77, a78, a79, a80, a81, a82, a83 };
207 PidObject *a85 = &a84;
208 real a87 = 0.0;
209 real a88 = 0.0;
210 real a89 = 0.0;
211 real a90 = 0.0;
212 real a91 = 0.0;
213 real a92 = 0.0;
214 real a93 = 0.0;
215 real a94 = 0.0;
216 real a95 = 0.0;
217 real a96 = 0.0;
218 real a97 = 0.0;
219 real a98 = 0.0;
220 real a99 = 0.0;
221 real a100 = 0.0;
222 PidObject a101 = { a87, a88, a89, a90, a91, a92, a93, a94, a95, a96, a97, a98, a99, a100 };
223 PidObject *a102 = &a101;
224 real a104 = 0.0;
225 real a105 = 0.0;
226 real a106 = 0.0;
227 real a107 = 0.0;
228 real a108 = 0.0;
229 real a109 = 0.0;
230 real a110 = 0.0;
231 real a111 = 0.0;
232 real a112 = 0.0;
233 real a113 = 0.0;
234 real a114 = 0.0;
235 real a115 = 0.0;
236 real a116 = 0.0;
237 real a117 = 0.0;
238 PidObject a118 = { a104, a105, a106, a107, a108, a109, a110, a111, a112, a113, a114, a115, a116, a117 };
239 PidObject *a119 = &a118;
240 real a121 = 0.0;
241 real a122 = 0.0;
242 real a123 = 0.0;
243 real a124 = 0.0;
244 real a125 = 0.0;
245 real a126 = 0.0;
246 real a127 = 0.0;
247 real a128 = 0.0;
248 real a129 = 0.0;
249 real a130 = 0.0;
250 real a131 = 0.0;
251 real a132 = 0.0;
252 real a133 = 0.0;
253 real a134 = 0.0;
254 PidObject a135 = { a121, a122, a123, a124, a125, a126, a127, a128, a129, a130, a131, a132, a133, a134 };
255 PidObject *a136 = &a135;
256 real a138 = 0.0;
257 real a139 = 0.0;
258 real a140 = 0.0;
259 real a141 = 0.0;
260 real a142 = 0.0;
261 real a143 = 0.0;
```

```
262     real a144 = 0.0;
263     real a145 = 0.0;
264     real a146 = 0.0;
265     real a147 = 0.0;
266     real a148 = 0.0;
267     real a149 = 0.0;
268     real a150 = 0.0;
269     real a151 = 0.0;
270     PidObject a152 = { a138, a139, a140, a141, a142, a143, a144, a145, a146, a147, a148, a149, a150, a151 };
271     PidObject *a153 = &a152;
272     real a155 = 0.0;
273     real a156 = 0.0;
274     real a157 = 0.0;
275     real a158 = 0.0;
276     real a159 = 0.0;
277     real a160 = 0.0;
278     real a161 = 0.0;
279     real a162 = 0.0;
280     real a163 = 0.0;
281     real a164 = 0.0;
282     real a165 = 0.0;
283     real a166 = 0.0;
284     real a167 = 0.0;
285     real a168 = 0.0;
286     PidObject a169 = { a155, a156, a157, a158, a159, a160, a161, a162, a163, a164, a165, a166, a167, a168 };
287     PidObject *a170 = &a169;
288     PidObject *a178[6];
289     a178[0] = a85;
290     a178[1] = a102;
291     a178[2] = a119;
292     a178[3] = a136;
293     a178[4] = a153;
294     a178[5] = a170;
295     controllerInit ( a178 );
296     int a180 = 0;
297     int a181 = 3;
298     cf_lib_vTaskSetApplicationTaskTag ( a180, a181 );
299     systemWaitStart ( );
300     int a182 = cf_lib_xTaskGetTickCount ( );
301     loop_start_label18: ;
302     goto label19;
303     label19: ;
304     int a185 = 2;
305     int a183 = cf_lib_vTaskDelayUntil ( a182, a185 );
306     a182 = a183;
307     cf_lib_imu9Read ( a5, a11, a17 );
308     bool a189 = imu6IsCalibrated ( );
309     bool a190 = true;
310     if ( a189 == a190 ) { goto label20; };
311     goto label21;
312     label20: ;
313     commanderGetRPY ( a29, a32, a35 );
314     cf_lib_commanderGetRPYType ( a47, a50, a53 );
315     int a198 = 1;
316     int a199 = a67 + a198;
317     a67 = a199;
```

```
318     int a201 = 2;
319     if ( a199 < a201 ) { goto label22; };
320     real a203 = 1.0;
321     real a204 = 500.0;
322     real a205 = 2.0;
323     real a206 = a204 / a205;
324     real a207 = a203 / a206;
325     cf_lib_sensfusion6UpdateQ ( a5, a11, a207 );
326     sensfusion6GetEulerRPY ( a20, a23, a26 );
327     real a215 = *a20;
328     real a217 = *a23;
329     real a219 = *a26;
330     real a221 = *a29;
331     real a223 = *a32;
332     real a225 = *a35;
333     real a226 = -a225;
334     controllerCorrectAttitudePID ( a215, a217, a219, a221, a223, a226, a38, a41, a44, a136, a153, a170 );
335     int a233 = 0;
336     a67 = a233;
337     label22: ;
338     cf_lib_LHS_Equals_Neg_RHS ( a44, a35 );
339     real * a237 = a5;
340     real a239 = *a38;
341     real a241 = *a41;
342     real a243 = *a44;
343     controllerCorrectRatePID ( a237, a239, a241, a243, a59, a62, a65, a85, a102, a119 );
344     cf_lib_commanderGetThrust ( a56 );
345     int a252 = *a56;
346     int a253 = 0;
347     if ( a252 <= a253 ) { goto label23; };
348     int a255 = *a56;
349     int a257 = *a59;
350     int a259 = *a62;
351     int a261 = *a65;
352     int a262 = -a261;
353     distributePower ( a255, a257, a259, a262 );
354     goto label21;
355     label23: ;
356     int a263 = 0;
357     int a264 = 0;
358     int a265 = 0;
359     int a266 = 0;
360     distributePower ( a263, a264, a265, a266 );
361     controllerResetAllPID ( a178 );
362     label21: ;
363     goto loop_start_label18;
364
365     return;
366   }
367
368   void distributePower ( int a0, int a1, int a2, int a3 ){
369     int a8 = a0 + a2;
370     int a10 = a8 + a3;
371     int a5 = limitThrust ( a10 );
372     int a15 = a0 - a1;
373     int a17 = a15 - a3;
```

```
374     int a12 = limitThrust ( a17 );
375     int a22 = a0 - a2;
376     int a24 = a22 + a3;
377     int a19 = limitThrust ( a24 );
378     int a29 = a0 + a1;
379     int a31 = a29 - a3;
380     int a26 = limitThrust ( a31 );
381     int a32 = 0;
382     cf_lib_motorsSetRatio ( a32, a5 );
383     int a34 = 1;
384     cf_lib_motorsSetRatio ( a34, a12 );
385     int a36 = 2;
386     cf_lib_motorsSetRatio ( a36, a19 );
387     int a38 = 3;
388     cf_lib_motorsSetRatio ( a38, a26 );
389     return;
390   }
391
392   int limitThrust ( int a0 ){
393     int a4 = 65535;
394     if ( a0 <= a4 ) { goto label24; };
395     a0 = a4;
396     goto label25;
397     label24: ;
398     int a9 = 0;
399     if ( a0 >= a9 ) { goto label25; };
400     int a10 = 0;
401     a0 = a10;
402     label25: ;
403     return a0;
404   }
```

Bibliography

[1] M. Weiser, "Ubiquitious computing," *Computer*, vol. 26, no. 10, pp. 71–72, 1993.

[2] H. Sundmaeker, P. Guillemin, P. Friess, and S. Woelfflé, *Vision and challenges for realising the Internet of Things*. EUR-OP, 2010.

[3] J. Sousanis, "World vehicle population tops 1 billion units." `http://wardsauto.com/ar/world_vehicle_population_110815`, August 2011. [Online; accessed Sept 2014].

[4] M. Broy, I. H. Kruger, A. Pretschner, and C. Salzmann, "Engineering automotive software," *Proceedings of the IEEE*, vol. 95, no. 2, pp. 356–373, 2007.

[5] D. L. Dvorak *et al.*, "Nasa study on flight software complexity," *NASA office of chief engineer*, 2009.

[6] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[7] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[8] D. Bandyopadhyay and J. Sen, "Internet of things: Applications and challenges in technology and standardization," *Wireless Personal Communications*, vol. 58, no. 1, pp. 49–69, 2011.

[9] M. Friedewald and O. Raabe, "Ubiquitous computing: An overview of technology impacts," *Telematics and Informatics*, vol. 28, no. 2, pp. 55–65, 2011.

[10] N. G. Leveson and C. S. Turner, "An investigation of the Therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, 1993.

[11] J.-L. Lions *et al.*, "Ariane 5 Flight 501 Failure, report by the enquiry board," tech. rep., CNES, Paris, 1996.

[12] N. Engineering and S. Center, "Technical support to the NHTSA on the reported Toyota Motor Corporation unintended acceleration investigation.," tech. rep., NASA, 2011.

[13] M. Blackburn, R. Busser, A. Nauman, R. Knickerbocker, and R. Kasuda, "Mars polar lander fault identification using model-based testing," in *Software Engineering Workshop, 2001. Proceedings. 26th Annual NASA Goddard*, pp. 128–135, IEEE, 2001.

[14] J. S. Gourlay, "A mathematical framework for the investigation of testing," *Software Engineering, IEEE Transactions on*, no. 6, pp. 686–709, 1983.

[15] J.-R. Abrial, *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[16] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin, "Rodin: an open toolset for modelling and reasoning in Event-B," *International journal on software tools for technology transfer*, vol. 12, no. 6, pp. 447–466, 2010.

[17] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[18] K. Stobie, "Too darned big to test," *Queue*, vol. 3, no. 1, pp. 30–37, 2005.

[19] M.-C. Gaudel, "Testing can be formal, too," in *TAPSOFT'95: Theory and Practice of Software Development*, pp. 82–96, Springer, 1995.

[20] J. Tretmans, "Model based testing with labelled transition systems," in *Formal methods and testing*, pp. 1–38, Springer, 2008.

[21] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2010.

[22] J.-R. Abrial, "From z to b and then event-b: Assigning proofs to meaningful programs," in *Integrated Formal Methods*, pp. 1–15, Springer, 2013.

[23] A. Romanovsky and M. Thomas, *Industrial deployment of system engineering methods*. Springer, 2013.

[24] S. D. Chowdhury, "Strategic roads that diverge or converge: GM and Toyota in the battle for the top," *Business Horizons*, vol. 57, no. 1, pp. 127–136, 2014.

[25] B. Foote and J. Yoder, "Big ball of mud," *Pattern languages of program design*, vol. 4, pp. 654–692, 1997.

[26] B. Vlasic and M. Apuzzo, "Toyota is fined $1.2 billion for concealing safety defects." `http://www.nytimes.com/2014/03/20/business/toyota-reaches-1-2-billion-settlement-in-criminal-inquiry.html?_r=0`, March 2014. [Online; accessed Sept 2014].

[27] J. Selna, "Toyota Motor Corp. Unintended acceleration marketing, sales practices, and products liability litigation," July 2013. Case No. 8:10ML 02151 JVS (FMOx).

[28] Q. Li and C. Yao, *Real-time concepts for embedded systems*. CRC Press, 2003.

[29] D. Pearce, *Whiley, A Programming Language with Extended Static Checking*. Victoria University, Wellington, 2014.

[30] D. Pearce, *Getting Started with Whiley*. Victoria University, Wellington, 2014.

[31] D. Pearce, *The Whiley Language Specification*. Victoria University, Wellington, 2014.

[32] J. C. King, "A program verifier.," tech. rep., DTIC Document, 1969.

[33] T. Hoare, "The verifying compiler: A grand challenge for computing research," in *Modular Programming Languages*, pp. 25–35, Springer, 2003.

[34] S. M. Loos, A. Platzer, and L. Nistor, "Adaptive cruise control: Hybrid, distributed, and now formally verified," in *FM 2011: Formal Methods*, pp. 42–56, Springer, 2011.

[35] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[36] K. Rustan and M. Leino, "Developing verified programs with Dafny," in *Proceedings of the 4th international conference on Verified Software: theories, tools, experiments*, pp. 82–82, Springer-Verlag, 2012.

[37] K. R. M. Leino, "Specification and verification of object-oriented software," *Engineering Methods and Tools for Software Safety and Security*, vol. 22, pp. 231–266, 2009.

[38] "Bitcraze and the crazyflie." `http://www.bitcraze.se/`. Accessed: March 2014.

[39] "Crazyflie kit electronics explained." `http://wiki.bitcraze.se/projects:crazyflie:hardware:explained`. Accessed: March 2014.

[40] "Arm The Architecture for the Digital World." `http://www.arm.com/`. Accessed: March 2014.

[41] "St life.augmented." `http://www.st.com/web/en/home.html`. Accessed: March 2014.

[42] "Firmware for the crazyflie nano quadcopter." `https://github.com/bitcraze/crazyflie-firmware`. Accessed: March 2014.

[43] S. Skogestad, "Simple analytic rules for model reduction and pid controller tuning," *Journal of process control*, vol. 13, no. 4, pp. 291–309, 2003.

[44] D. M. Ritchie, "The development of the C language," *ACM SIGPLAN Notices*, vol. 28, no. 3, pp. 201–208, 1993.

[45] H. Schildt, *C/C++ programmer's reference*. McGraw-Hill, Inc., 2000.

[46] ISO, *C90 Standard*, 1990. ISO/IEC 9899:1990.

[47] ISO/IEC JTC1 SC22 WG14 N1169, "Programming languages - C - Extensions to support embedded processors," Tech. Rep. TR 18037, ISO/IEC, Geneva, Switzerland, 2006.

[48] "FreeRTOS- Real Time Operating System." `http://www.freertos.org/`. Accessed: June 2014.

[49] C. Svec, "The architecture of open source applications, vol II - FreeRTOS." `http://aosabook.org/en/freertos.html`. Accessed: May 2014.

[50] H. Kopetz, *Real-time systems: Design principles for distributed embedded applications*. Springer, 2011.

[51] "FreeRTOS- ROM use." `http://www.freertos.org/FAQMem.html#ROMUse`. Accessed: June 2014.

[52] P.-H. Kamp, "malloc (3) revisited.," in *USENIX Annual Technical Conference*, 1998.

[53] P. Marwedel, *Embedded system design*, vol. 1. Springer, 2006.

[54] IEEE, *POSIX.1-2008*, 1003.1 ed., 2008. Accessed Sept 2014. `http://pubs.opengroup.org/onlinepubs/9699919799/toc.htm`.

[55] GNU, *The GNU C Library (glibc)*, September 2014. v2.20. Accessed Oct 2014. `https://www.gnu.org/software/libc/index.html`.

[56] M. Ruarus, "Compiling Whiley programs for a general purpose GPU," tech. rep., Victoria University, Wellington, 2013.

[57] R. C. Waters, "Program translation via abstraction and reimplementation," *Software Engineering, IEEE Transactions on*, vol. 14, no. 8, pp. 1207–1228, 1988.

[58] M. P. Ward, "Pigs from sausages? Reengineering from Assembler to C via FermaT transformations," *Science of Computer Programming*, vol. 52, no. 1, pp. 213–255, 2004.

[59] M. Ward, "Assembler restructuring in FermaT," in *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pp. 147–156, IEEE, 2013.

[60] M. P. Ward, "Assembler to C migration using the FermaT transformation system," in *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pp. 67–76, IEEE, 1999.

[61] Digital Research, INC., Pacific Grove, California, *XLT86, 8080 to 8086 Assembly Language Translator, USER'S GUIDE*, September 1981.

[62] "GCC, the GNU Compiler Collection." `http://www.gnu.org/software/gcc/`. Accessed: March 2014.

[63] "GLib Reference Manual." `http://www.gnu.org/software/gcc/`. Accessed: Oct 2014.

[64] N. Stollon, "JTAG use in debug," in *On-Chip Instrumentation*, pp. 31–48, Springer, 2011.

[65] A. Hopkins and K. McDonald-Maier, "Debug support for complex systems on-chip: A review," *IEE Proceedings-Computers and Digital Techniques*, vol. 153, no. 4, pp. 197–207, 2006.

[66] F. J. Massey Jr, "The kolmogorov-smirnov test for goodness of fit," *Journal of the American statistical Association*, vol. 46, no. 253, pp. 68–78, 1951.

[67] D. Svoboda and L. Wrage, "Pointer ownership model," in *System Sciences (HICSS), 2014 47th Hawaii International Conference on*, pp. 5090–5099, IEEE, 2014.