Formal Verification of a Real-Time Operating System

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Gadi de Leon Tellez Espinosa

Copyright Gadi de Leon Tellez Espinosa, August, 2012. All Rights Reserved.

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

Errors caused by the interaction of computer systems with the physical world are hard to mitigate but errors related to the underlying software can be prevented by a more rigorous development of software code. In the context of critical systems, a failure caused by software errors could lead to consequences that are determined to be unacceptable. At the heart of a critical system, a real-time operating system is commonly found. Since the reliability of the entire system depends upon having a reliable operating system, verifying that the operating systems functions as desired is of prime interest. One solution to verify the correctness of significant properties of an existing real-time operating system microkernel (FreeRTOS) applies assisted proof checking to its formalized specification description. The experiment consists of describing real-time operating system characteristics, such as memory safety and scheduler determinism, in Separation Logic — a formal language that allows reasoning about the behaviour of the system in terms of preconditions and postconditions. Once the desired properties are defined in a formal language, a theorem can be constructed to describe the validity of such formula for the given FreeRTOS implementation. Then, by using the Coq proof assistant, a machine-checked proof that such properties hold for FreeRTOS can be carried out. By expressing safety and deterministic properties of an existing real-time operating systems and proving them correct we demonstrate that the current state-of-the-art in theorem-based formal verification, including appropriate logics and proof assistants, make it possible to provide a machine-checked proof of the specification of significant properties for FreeRTOS.

Acknowledgements

I would like to thank my dad, mom and extended family for all their support and words of encouragement throughout my life. Thank you for helping me fulfill my dreams.

This thesis would not have been possible without my supervisor Prof. Christopher Dutchyn who guided me in pursuing a topic of my interest, even when this required a few adjustments along the way. Chris has been a great mentor, teacher and advisor. Thank you for all the inspiring conversations.

This thesis would not have been the same without the help from the members of my thesis committee: Prof. Michael Horsch, Prof. Dwight Makaroff, and Prof. Ronald Bolton. Thank you for your guidance and valuable feedback to complete this written work.

Finally, I would like to thank all my dear friends. Thanks for making life this much fun.

To my beloved family...

Contents

List of Tables

List of Figures

List of Abbreviations

AST	Abstract Sytax Tree
CIC	Calculus of Inductive Constructions
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
FIFO	First-In First-Out
LOC	Lines of Code
OS	Operating System
RTOS	Real Time Operating System
VCG	Verification Condition Generator

Chapter 1

Introduction

Errors in computer systems are very common [24]. Due to the nature of software systems and their relation with the physical world, the current assumption is that the elimination of all errors is intractable [25]. This assumption is supported by the reasonable view expressed by Neufelder [42] in which a computer system is a biological system in the sense that the software program is not isolated but it is integrated with the hardware, human operators and the physical world. Since it is impossible to prevent unplanned events in the physical world such as natural disasters, this view considers it impossible to develop an error-free computer system.

The presence of errors in computer software raises the ongoing concern that the system might fail to behave as expected, resulting in unforeseen and, in some cases, catastrophic consequences. When software systems present the risk of potential catastrophic consequences, such as placing lives, property, or the environment at risk, the accepted rate of mishaps[1] is desired to be virtually nonexistent.

Hundreds of incidents caused by the use of computer systems and related technology have taken place for over 30 years [44, 45]. While many of these incidents were caused by common, uncontrolled events of the physical world such as hardware failure and natural disasters, there are many others for which the primary source of errors was the software [43]. As an example of the high costs incurred by a software system that failed because of software errors, consider the case of Therac25, a computer system intended to apply therapeutic radiation for chemotherapy but which inadvertently killed and maimed patients before being forced off the market [21]. Six known accidents involved massive overdoses by the Therac-25 with resulting deaths and serious injuries in what was described as the worst series of radiation accidents in the 35-year history of medical accelerators [37].

Errors caused by the interaction of computer systems with the physical world are certainly hard to mitigate; but, this does not stop software engineers from focusing on errors related to the software program. These errors can possibly be prevented by a more detailed and rigorous development of software code. Experience suggests that errors in a software program are most often caused by design and implementation

[1] The US Department of Defense defines mishap as an unplanned event or series of events that result in death, injury, occupational illness, damage to or loss of equipment or property, or damage to the environment.

faults [26] while only a few are caused by incorrect system operation and use. Design faults occur when a designer either misunderstands a specification or simply the specification is not well-described, causing an unexpected, and frequently incorrect, program behaviour. Implementation faults are the result of erroneous code that does not meet the system specification. It is estimated that between 60% and 90% of computer errors are the result of software design and implementation faults [26].

A common approach to identify and eliminate errors in software programs is testing. Running functional tests on a piece of software consists of examining the behaviour of the system under as many scenarios as the tester can devise. The problem with this approach is the limitation on the number of scenarios that humans can specify and implement. Despite the efforts of making the list of test cases as exhaustive as possible and the use of techniques such as random testing and simulation, exhaustive testing is impossible to achieve due to the impossibly large number of scenarios that need to be tested in a system.

Testing certainly reduces the number of errors in a software program but it does not guarantee the elimination of all design and implementation errors due to the impossibility to achieve exhaustive testing. Because of this, testing could be a good solution for those systems that present a low cost of impact in the presence of errors. On the other hand, this approach is not appropriate for critical systems — those for which a failure could lead to consequences that are determined to be unacceptable [34].

Critical systems include, but are not limited to, real-time systems. Within these systems, the operating system is, arguably, the heart of the software program because the reliability of the entire system depends upon having a reliable operating system [17]. Due to the importance of the operating system on the correctness of a real time system, the operating system must be guaranteed to have a correct implementation in order for the system to be considered reliable — a necessary but not sufficient condition.

Providing evidence of the correct implementation of computer software involves showing that the known and desired properties of the system specification are met by the given implementation. In a broad way, the correctness properties of any system can be classified in two categories: safety and liveness [35]. A safety property is one which states that something unexpected will not happen. An example of a safety property is the assurance that the result of a computation will not cause buffer overflows. A liveness property is one which states that something must happen. An example of a liveness property is that a program will terminate its execution.

In the context of a real-time operating system, a specific safety property often takes the shape of mutual exclusion; whereas, liveness properties are commonly required of the scheduler mechanism. Therefore, showing the correctness of a real-time operating system implementation requires evidence of its safety and liveness properties.

In this thesis, the proposed solution to proving the correctness of the necessary and significant properties of a real-time operating system, such as memory safety and scheduler liveness, consists of translating the description of such properties into a formal language and then providing evidence to show that these properties are true for the given implementation. Therefore, the first step is to be able to specify the safety and liveness properties of the real time operating system in a formal language that allows reasoning about the behaviour of the system. Once the formal description is available, a proof of whether the properties hold for the implementation or not can be checked using a proof assistant.

Following this methodology would demonstrate that the state of the art in theorem-based formal verification, including formal languages such as Separation Logic, and proof assistants such as Coq, enables a machine-checked proof of the significant properties of a real-time operating system. To this effect, first I will introduce the topics related to the formal description of the properties of a system and the assisted verification of its correct implementation. Next, the details of the experiment functioning as supporting evidence to the previous argument are presented in the experiment chapter. The experiment has been carried out using FreeRTOS [39], a C implementation of a real-time operating system, and Coq, a formal proof management system that provides the formal language to write theorems about the correct behaviour of FreeRTOS along with the environment for interactive development of machine-checked proofs. To describe the requirements to formally specify an operating system, the evaluation of the experiment, including an assessment of the quality of the tools used in its development, as well as a comparison of my approach with other related work is presented in the evaluation chapter. Finally, the conclusions drawn from this research project, the acknowledgement of its limitations, and proposed future work in the summary chapter bring this thesis to a close.

Chapter 2

Background

To appreciate the process of applying state-of-the-art formal verification to a production real-time operating system there are three basic requirements

1. An understanding of what constitutes a real-time operating in terms of the properties that define it.
2. An understanding of the potential languages in which to express these properties in a formal way.
3. An appreciation of the different approaches to formal verification to identify the most appropriate solution to this problem.

The discussion of these topics is the subject of this chapter.

2.1 Real-Time Operating Systems (RTOS)

First we start with clearly describing an Operating System (OS). An operating system is the piece of software whose job is "to provide user programs with a better, simpler, cleaner model of the computer and to handle the resources the system is made up of such as processors and memory" [52]. Depending on the purpose for what the system is designed and the underlying hardware architecture, there exist a variety of operating systems ranging from mainframe operating systems — which are heavily oriented toward processing many jobs at once — personal computer operating systems, embedded operating systems, real-time operating systems, to smart card operating systems — the smallest operating system designed to run on credit card-sized devices containing a CPU chip.

A system is considered safety-critical when the possibility of a failure carries a high probability of causing death, injury and damage to equipment, property and the environment. Examples of safety-critical systems span several domains of science and technology including air traffic control systems, defense and space systems, embedded automotive electronics, health and safety systems, and industrial process controllers.

Typically, a safety-critical system consists of a *controlled system*, which can be viewed as the environment with which the system interacts, and a *controlling system*. For example, in an automated factory, the controlled system is the factory floor with its robots, assembling stations, and the assembled parts; whereas the controlling system is the computer and human interfaces that manage and coordinate the activities on the factory floor [50].

At the heart of the controlling system is found an operating system in charge of managing the computer resources. Due to the deterministic timing requirements of some safety-critical systems, the operating system has to be designed for real-time applications. In this kind of applications, the correctness of the system behaviour depends not only on the logical results of the computation but also on the time these results are produced [50]; real-time operating systems are, therefore, commonly used in safety-critical systems.

As any other operating system, a real-time operating system is in charge of providing basic support for scheduling, resource management, task synchronization, and communication. What makes a real-time operating system different from other OS is that it must also provide facilities that guarantee precise timing to ensure the required computation is carried out within the stated deadline [51].

Based on levels of determinism over the timing of a given computation, real-time systems are classified as *soft* real-time systems versus *hard* real-time systems. In a soft real-time system, the results of an otherwise valid computation might be considered useless or invalid in the case where the response time is not within the requirements. On the other hand, failing to meet the response time of hard real-time system can result in catastrophic consequences.

The complexity of the controlling system in a safety-critical system spans from very simple micro controllers to complex and distributed systems [50]. Despite this complexity, it is often the case that the entire controlling system is supported by stripped down and optimized version of RTOS kernels, commonly known as *microkernels* — arguably the most critical part of any computer system [17].

The basic idea behind the microkernel design is to achieve high reliability by splitting the operating system up into small modules with a well-defined interface that serves as the connection to the rest of the system implementation as processes running on top of the microkernel. The decision of what modules to include in the microkernel varies among different implementations. Nonetheless, without an existing a standard, some experts agree on a microkernel as the core of the operating system that provides process scheduling, memory management, and communication services [17, 22, 52]. These critical modules are the closest software layer sitting on top of the hardware and serve as an interface to other required modules, which depending on the application could include device drivers, file systems and network protocols. These modules sitting on top of the microkernel serve in turn as the interface to serve user level software. This strict layered approach can be seen in figure 2.1.

The kernel is therefore not only a significant piece of software in its own right, but also a critical module in a safety-critical system. Under these premises, verifying a real-time operating system kernel to provide a guarantee of its implementation correctness is a fundamental priority in characterizing a safety-critical system as reliable.

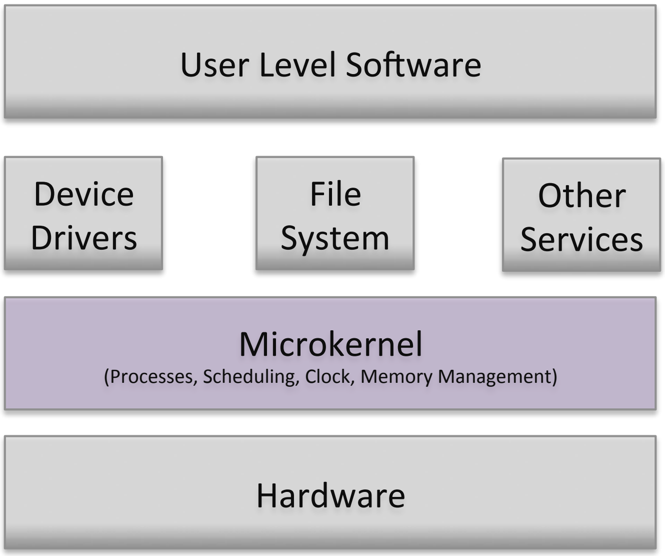

Figure 2.1: Microkernel Structure

2.1.1 FreeRTOS

FreeRTOS [39] is a portable, open-source real-time microkernel with more than 77,500 downloads per year that attest for its popularity in the implementation of real-time systems. FreeRTOS is a professional-grade implementation of a real-time operating system written in the C programming language. Its implementation is contained in three files comprising 2,200 lines of code plus an extra file with the specifics for the chosen target architecture on which the system will be deployed. FreeRTOS supports a total of 31 architectures, including x86, PIC and ARM, among others.

With only six to ten kilobytes of footprint, FreeRTOS is specially designed for its implementation in small, embedded real-time systems. To this end, FreeRTOS contains one primary primitive, the message queue, which is then used as a base for all other primitives. The design decision of providing only the modules for memory management and task management removes the requirement for many separate software modules and their corresponding additional footprint.

In a way, FreeRTOS pays the price of its high portability by its lack of readiness to be deployed as a standalone system. In other words, FreeRTOS is not a full-featured user-level operating system but it is better described as a microkernel that provides the core real-time scheduling, inter-task communication, timing and memory management functionality. The rest of the functionality required of a full real-time system implementation such as console interface, device drivers and networking stacks as are left for the programmer of the final implementation to implement. FreeRTOS is designed so that real-time applications can be structured as a set of independent tasks. Each task executes within its own context with no dependency on other tasks within the system or the scheduler itself [39].

Task Management A task constitutes the basic computation unit in FreeRTOS. A task consists of

1. A state - depending on whether the task is *running*, *suspended*, *blocked*, or *ready* to execute.
2. A priority - an integer value ranging from zero up to a maximum priority value define at compile time
3. An execution context - storing the call stack and the register values when the task is not executing

FreeRTOS scheduler is based on priorities. During the selection of the next executing task, the scheduler always selects the task with the highest priority among the *ready* and *running* tasks. In order to keep to system running, even when there are no tasks ready to be executed, FreeRTOS automatically creates an *idle task* with priority zero that is executed whenever there are no other tasks that require processing time.

A peculiar characteristic of FreeRTOS that undermines its application for systems with dynamic creation of tasks is its lack of a schedulability algorithm. FreeRTOS provides the capability of creating tasks dynamically (i.e. during execution time) and sets no restrictions on the number of tasks that can be created but does not perform an analysis to determine whether or not the system will be able to meet its deadlines based on the number of tasks that the OS is serving. Therefore, FreeRTOS is a suitable solution for systems that deal with a static number of tasks since the programmer can perform the schedulability algorithm before the system is deployed but a better solution is needed when dynamic creation of tasks is required of the system to be implemented[1]. Systems that do not require dynamic creation of tasks are the focus of this thesis.

Memory Management FreeRTOS provides a simple but useful memory management algorithm that satisfies the requirements of the majority of applications in which FreeRTOS is used [39]. The abstract model of the memory consists of a single array structure that is subdivided as blocks of memory are requested from the tasks running on top of the OS. The total size of the array (i.e. the total size of the heap) is determined during compilation time.

A distinctive characteristic of FreeRTOS memory manager is the impossibility of freeing memory locations once they have been allocated. The argument behind this decision according to FreeRTOS' developers is that "the majority of deeply embedded applications create all the elements required when the system boots, and then use all of these elements for the lifetime of the program execution" [39]. A great advantage of this assumption is that the implementation of the memory manager always takes the same amount of time to serve a memory request.

[1] A potential solution is the implementation of the schedulability algorithm as a task running on top of FreeRTOS microkernel that gets executed every time a task is created.

2.2 Formal Verification

Next, we turn our attention to formal verification. The notion of being able to provide a proof that a program is correct with respect to a specification has been around for over 40 years [7]. During this time, formal verification has developed into two fundamentally different approaches: model checking and theorem proving. Each of these is discussed in the remainder of this section.

2.2.1 Model Checking

Model checking is characterized by building an abstraction of all possible states that a computer program can reach, usually represented as a state machine. Once this model has been constructed, one can analyze its structure and verify its properties, checking that none of the states present in this model are undesirable. Although there are many advantages for model checking systems, such as

1. Being able to reason about concurrency thanks to the development of temporal logic

2. The ability to produce a counterexample execution trace in case the specification is not satisfied

3. The high level of automation of the tools used in model checking

there is a major caveat to it: the state explosion problem [14]. The number of states a program can reach during execution grows exponentially by different factors like the number of variables and components that constitute the system. In an effort to eliminate the state explosion problem, an abstraction technique is used to reduce the size of the state space that must be searched. Thanks to this abstraction, model checking has become tractable and has been successfully applied to find errors in hardware controllers [3, 23]. Nevertheless, special attention is required when abstracting the state space complexity. Due to this abstraction, it is possible to build a false positive model — a model incorrectly categorized as correct — by not considering special states, making this solution prone to errors.

An example of the successful implementation of model checking technology in the verification of hardware and software is Intel, whose pioneering work in applying model checking to build industrial verified systems has been present since 1990 [23]. Since the early beginnings, Intel has reported a successful usage of such verification technology in the discovery of bugs that either would have been found much later in the design cycle or bugs that might otherwise have escaped all the verification tools.

The first generation of Intel's verification tools focused on verifying hardware designs. Two lead CPU design teams used the verification tools on selected, high-risk areas in which new complex functionality was added to real, sophisticated design projects. Properties were developed to capture the intended relation between the inputs and the outputs of each module. Properties on the outputs of a module were verified using the assumptions on the inputs of the module. Properties on the output signals of the module served also as assumptions on the inputs of the next module. Despite the challenges faced along the way, the added value that verification tools provided on the design and implementation of complex hardware lead to the implementation of formal verification using model checking to other areas and projects.

Model checking is not limited to the verification of hardware; throughout the years, Intel has also developed a variety of specification languages, specification coverage tools, model checkers and simulation engines for the verification of software. MicroFormal [2] — a technology for fully automated formal verification of functional backward compatibility of microcode — is a recent example of the successful implementation of model checking for software verification. The microprograms being verified using MicroFormal are quite complex CISC flows that involve both memory interaction, and multiple sanity checks that can result in exceptions. In a verification session, microcode of the new generation is compared against the old generation. The novel technology provided by MicroFormal has been recognized as one that can significantly improve the quality of microcode.

In spite of the success of model checking-based verification for smaller systems[2], better solutions are needed to handle large designs of bigger complexity. For this reason, model checking is not a tractable solution when proving the properties of an operating system.

2.2.2 Theorem Proving

An alternative technology for verifying software is theorem proving. Theorem proving is characterized by describing the specification of a computer program in a formal reasoning system, which consists of a set of axioms and a set of inference rules that can be used to derive new theorems. Verifying a property amounts to generating a proof, where each deduction step in the proof is an instance of one of the inference rules applied to axioms or previously proved theorems [7]. The remainder of this section provides more detail and uses notation from sequent calculus and proof trees from [10].

Due to the complexity of a software specification, it is common to deal with individually independent properties of the system as separate *judgements*. Once the specification has been constructed, one can verify that each of the properties, or judgements, hold true using deductive reasoning. If, given a set of axioms, one can construct a *derivation tree* following the *inference rules* defined by the meta-logic in which the specification has been written, then the general validity of the formula has been proven to hold true.

In theorem-based formal verification, a proof system consists of a collection of inference rules of the form:

$$\frac{J_1 \qquad ... \qquad J_k}{J}\ \{\text{name}\}.$$

The judgements above the horizontal line are called the *premises* of the rule, and the judgement below the line is called its *conclusion*. The inference rule states that the premises are sufficient for the conclusion:

[2] Given the number of variables and instructions involved in the execution of the program, the complexity of the microcode verified by Intel is less than the complexity of a real-time operating system.

to show J, it is enough to show $J_1, ..., J_k$. If a rule has no premises, then the rule is called an *axiom*. An axiom is, therefore, a rule stating that its conclusion holds unconditionally.

A derivation tree (or a proof) of a judgement is a finite composition of rules, starting with axioms and ending with the desired judgement to be proven (also called the theorem). In a derivation tree, each node is a rule whose children are derivations of its premises.

For example, consider a reasoning system for natural numbers where the number 0 is unconditionally considered to be a natural number. Formally, we state this as the axiom:

$$\frac{}{0\%\mathrm{nat}}$$

Moreover, consider a rule of inference to produce more natural numbers by stating that given a natural number, the successor of this number is also a natural number. Formally, we define this as:

$$\frac{n\%\mathrm{nat}}{\mathrm{s}(n)\%\mathrm{nat}}\ \text{successor}$$

Then, to provide a proof that the successor of the successor of the number 0, namely s(s(n)), is also a natural number, we construct a derivation tree starting with the unconditional judgement that 0 is a number and applying the successor rule of inference twice, ending with the judgment to be proven. Formally:

$$\cfrac{\cfrac{\cfrac{}{0\%\mathrm{nat}}}{\mathrm{s}(n)\%\mathrm{nat}}\ \text{successor}}{\mathrm{s}(\mathrm{s}(n))\%\mathrm{nat}}\ \text{successor}$$

There are two different approaches taken when building a derivation tree: *forward chaining* or *bottom-up*, and *backward-chaining* or *top-down* [28].

The bottom-up approach starts with the axioms and works forward towards the desired conclusion. More precisely, this approach maintains a set of derivable judgements and continually extends it by adding the conclusions of any rule whose premises are all present in the set. Bottom-up is undirected in the sense that it does not take into account the end goal when deciding how to proceed at each step. For this reason, bottom-up is usually used in automated provers, where finding a proof is attempted by an exhaustive application of inference rules to previously proven theorems and axioms.

The top-down approach starts with the desired conclusion and and works backwards towards axioms or other known results. More precisely, this approach maintains a queue of current goals (judgements) whose derivations are to be found. Initially, this queue contains only the property we wish to derive. At each stage the judgement is removed from the queue by providing a rule whose conclusion is that judgement. For each rule, the premises are added to the back of the queue and repeat the process until the queue is empty. This approach is more intuitive to the user, and it is often preferred in interactive provers. From now on, this thesis will only consider top-down approach when building derivation trees.

A theorem-based formal verification approach, hence, requires a definition of the logic that will determine the inference rules under which one can reason about the system and the properties to be proven. Such logic has to be expressive enough so that it can describe the system's specification in the same way natural languages does but with a higher level of rigour when compared to the latter so that it can be used in the derivation of formal proofs.

2.2.3 Hoare Logic

In 1969, Charles Hoare introduced an axiomatic method for proving that a computer program is *partially correct* with respect to a formal specification. A partial correctness specification holds only when the program terminates, but it is not valid otherwise. Alternately, *total correctness* guarantees that the program will terminate and the specification will be met at all stages of program execution.

Proposing that, as an exact science, computer programming could be subject to mathematical investigation, Hoare formulated a set of axioms and rules of inference which can be used in proofs of the properties of computer programs, later to be known as Hoare Logic [29]. Hoare Logic states that the intended function of a program can be specified my making general assertions about the *computational state* of a program before and after its execution. A computational state is defined by a *store* that maps variable names into values. With the purpose of expressing the aforementioned assertions in a formal language, Hoare introduced the following notation, known as Hoare Triple:

$$\{P\}\ \mathtt{C}\ \{Q\}$$

where P is the required precondition for the program C to be executed and Q is the resulting state following the execution of C.

The $\{P\}$ C $\{Q\}$ triple is interpreted as follows: if assertion P is true before the execution of the program C, and such execution terminates, then the assertion Q will be true on its completion. The correctness of a program is then reduced to reasoning about individual statements in a program by introducing assertions surrounding each statement (i.e. subprograms of the main program).

$\{P\}\ \mathtt{skip}\ \{P\}$	Skip axiom
$\{P[X \leftarrow E]\}\ \mathtt{X\ :=\ E}\ \{P\}$	Assignment axiom

Table 2.1: Axioms for Hoare Logic

Hoare Logic axioms: The set of axioms formulated by Hoare to describe the behaviour of a computer program consists of the skip axiom and the assignment axiom listed in table 2.1. The skip axioms means that whatever assertion is true before executing the program `skip` it will hold true after completing the skip action. The assignment axiom can be interpreted as: any assertion which is to be true of `X` after the assignment is made must also have been true of the expression `E` before the assignment is made. The syntax in the assignment axiom refers to substitution of the term `X` by the expression `E` in the assertion P (commonly found as $[X/E]P$ in other formal languages).

$\dfrac{\{P_1\}\ \mathtt{C}_1\ \{P_2\} \qquad \{P_2\}\ \mathtt{C}_2\ \{P_3\}}{\{P_1\}\ \mathtt{C}_1\mathtt{;}\mathtt{C}_2\ \{P_3\}}$	Composition rule
$\dfrac{\{P \wedge B\}\ \mathtt{C}_1\ \{Q\} \qquad \{P \wedge \neg B\}\ \mathtt{C}_2\ \{Q\}}{\{P\}\ \mathtt{if\ B\ then\ C}_1\ \mathtt{else\ C}_2\ \{Q\}}$	Conditional rule
$\dfrac{\{P \wedge B\}\ C\ \{P\}}{\{P\}\ \mathtt{while\ B\ do\ C\ end}\ \{P \wedge \neg B\}}$	While rule

Table 2.2: Rules of Inference for Hoare Logic

Hoare Logic inference rules: The set of rules of inference formulated by Hoare to describe the behaviour of a computer program consists of the composition rule, the conditional rule and the while rule, as listed in table 2.2.

The *composition rule* states that if the postcondition of a program $\mathtt{C}_1$ is identical to the precondition of another program $\mathtt{C}_2$, then the execution of both programs in sequence will produce the results of the second program $\mathtt{C}_2$, provided that the precondition of the first program $\mathtt{C}_1$ is satisfied.

The *conditional rule* considers two programs $\mathtt{C}_1$ and $\mathtt{C}_2$ with the shared assertion Q as a resulting state of their execution. If the required preconditions are similar modulo the logical value of assertion B, then we can formalize the execution of either one of $\mathtt{C}_1$ or $\mathtt{C}_2$ under the logical value of B as a conditional statement.

The *while rule* supposes an assertion P which is true before and after the execution of program `C`. Furthermore, it is known a certain condition B which is true before executing the program `C` but happens to be false after such execution. This behaviour formalizes what happens when a program is executed a number of times determined by the condition B.

Hoare Logic structural rules: The *consequence rules* refer to the weakening of preconditions and strengthening of postconditions correspondingly. In other words, if P' is known to be a precondition for a program `C`, then so is any other assertion P which logically implies P'. That is, P is at least as strong as

$\dfrac{P \Rightarrow P' \qquad \{P'\}\,\mathtt{C}\,\{Q\}}{\{P\}\,\mathtt{C}\,\{Q\}}$	Consequence rule I
$\dfrac{\{P\}\,\mathtt{C}\,\{Q'\} \qquad Q' \Rightarrow Q}{\{P\}\,\mathtt{C}\,\{Q\}}$	Consequence rule II
$\dfrac{\{P\}\,\mathtt{C}\,\{Q\}}{\{P \wedge R\}\,\mathtt{C}\,\{Q \wedge R\}}$	Rule of constancy

Table 2.3: Structural Rules for Hoare Logic

P' so P' is weaker than P. Following the same logic, if Q' is known to be a postcondition for a program C, then so is any other assertion Q that follows logically from Q'. That is, Q' is at least as strong as Q.

The *rule of constancy* is vital for scalability. This rule permits the extension of a local specification of C involving only those variables actually used by the command with the addition of assertions about variables not modified by C. The rule of constancy requires the condition that no variable occurring free in R is modified by C. In other words, variables in the logic formula R that are not inside the scope of a quantifier are not modified by C.

The Hoare Logic deductive system is *sound*, meaning that its axioms and rules of inference prove only formulas that are valid with respect to the operational semantics of the language described by the formal system. In other words, If we prove the Hoare triplet $\{P\}\,\mathtt{C}\,\{Q\}$, then C is partially correct with respect to specification P, Q. [16].

On the other hand, Hoare Logic is not *complete* for the following reasons:

1. The partial correctness of a program C may not be expressible by a formula $\{P\}\,\mathtt{C}\,\{Q\}$. This is mainly because Hoare Logic relies on the expressive power of the underlying programming and assertion languages to which Hoare Logic is applied; hence, the expressive power of the assertion language may not be sufficient to describe the computational state that some programs require. For example, a program that computes addition by successive increments might not be provable in Hoare Logic if the underlying assertion language is based upon abacus arithmetic because addition is not expressible in this system [16]. In formal terms, the Hoare triple

$$\boxed{\{X = x \wedge Y = y\}} \quad \boxed{\begin{array}{l} \texttt{while}\neg(X = 0) \\ \texttt{do}(\\ \quad Y := \mathit{succ}(Y); \\ \quad X := \mathit{pred}(X) \\ ) \end{array}} \quad \boxed{\{Y = x \boxed{+} y\}}$$

is not expressible because the addition symbol used in the postcondtion cannot be represented in abacus arithmetic. To overcome this limitation, abacus arithmetic can be enriched to include the addition operation, resulting in Presburger Arithmetic. Nevertheless, Presburger Arithmetic has the same problem when dealing with the multiplication operation. Namely, the Hoare triple

$$\boxed{\{X = x \land Y = y\}} \quad \boxed{\begin{array}{l} Z := 0; \\ \texttt{while}\neg(X = 0) \\ \texttt{do}(\\ \quad Z := Z + Y; \\ \quad X := \mathit{pred}(X) \\ ) \end{array}} \quad \boxed{\{Z = x \boxed{*} y\}}$$

is not possible to construct because the multiplication symbol is not expressible in Presburger Arithmetic.

2. Some true formula $\{P\}$ `C` $\{Q\}$ may not be provable by Hoare Logic deductive system. This is of special attention when dealing with *while rule* of inference. It might be the case that given the assertions P and Q, the proof $\{P\}$ `while` B `do` C `end` $\{Q\}$ would involve an internal loop invariant which cannot be expressed by a formula of the underlying assertion language. For example, defining the following program with an underlying assertion language based on abacus arithmetic:

$$\boxed{\{X = x \land Y = 0\}} \quad \boxed{\begin{array}{l} \texttt{while}\neg(X = 0) \\ \texttt{do}(\\ \quad Y := \mathit{succ}(Y); \\ \quad X := \mathit{pred}(X) \\ ) \end{array}} \quad \boxed{\{Y = x\}}$$

The proof of this Hoare triple must involve the *while* rule with some invariant I such that $\{I \land \neg(X = 0)\}(Y := \mathit{succ}(Y); X := \mathit{pred}(X))\{I\}$ and the consequence rule so that $(P \Rightarrow I)$ and $(I \land (X = 0)) \Rightarrow Q$. Then, the invariant I stating that $(x = X \boxed{+} Y)$ is not expressible in this logic.

As important of a breakthrough as Hoare Logic was for program verification, it presents some limitations that make it untreatable for real life scenarios. Hoare himself foresaw this limitation when he mentioned "program proving, certainly at present, will be difficult even for programmers of high calibre; and may be applicable only to quite simple program designs." [29, p. 5] Arguably, the programmers could be trained in becoming experts in the use of Hoare Logic; nevertheless, the innate lack of expressiveness in Hoare Logic had to be dealt with by the elucidation of more expressive logics.

2.2.4 Separation Logic

Separation Logic is an extension of Hoare Logic that permits reasoning about low-level imperative programs that use shared mutable data structures, i.e. structures where updatable fields can be referenced from more than one point via more than one name or alias [48]. The resulting extension of Hoare Logic is the product of extending the expressiveness of the programming language which the logic describes. The extended imperative language adds new statements for the manipulation of mutable shared data structures, namely: allocation, lookup, mutation, and deallocation as shown in figure 2.2

$$
\begin{aligned}
\langle \mathit{statement} \rangle ::= {} & \ldots \\
& \mid \langle \mathit{var} \rangle := \mathbf{cons}(\langle \mathit{exp} \rangle, ..., \langle \mathit{exp} \rangle) && \text{(Allocation)} \\
& \mid \langle \mathit{var} \rangle := [\langle \mathit{exp} \rangle] && \text{(Lookup)} \\
& \mid [\langle \mathit{exp} \rangle] := \langle \mathit{exp} \rangle && \text{(Mutation)} \\
& \mid \mathbf{dispose} \langle \mathit{exp} \rangle && \text{(Deallocation)}
\end{aligned}
$$

Figure 2.2: Statements for the manipulation of mutable shared data structures.

The *allocation* statement activates as many heap cells as the number of parameters. The value of each cell is then initialized to the value to which each $\langle \mathit{exp} \rangle$ evaluates. It is important to notice that, aside from the requirement that the addresses of those cells be consecutive and previously inactive, the choice of addresses is indeterminate.

The *lookup* statement evaluates the expression $\langle \mathit{exp} \rangle$ and searches for the corresponding heap cell and assigns the value of such heap address to variable $\langle \mathit{var} \rangle$. To differentiate a heap location from a store location, the notation `"[]"` is adopted. If an inactive address is referenced by $\langle \mathit{exp} \rangle$, a memory fault is raised.

The *mutation* statement assigns the value of the right hand side $\langle \mathit{exp} \rangle$ to the heap cell reference to by the evaluation of the left hand side $\langle \mathit{exp} \rangle$. As with lookups, if an inactive address is referenced, a memory fault is raised.

Finally, the *dispose* statement deactivates the heap cell pointed to by the evaluation of the expression $\langle \mathit{exp} \rangle$. In case an inactive address is referenced by the expression, then a memory fault is raised.

Semantically, computational states are modified to contain not only a store but also a *heap*. Different from the store that maps names into values, a heap maps addresses to values. When a store and a heap are combined, usually the store maps names (i.e. variable names) into addresses, and then the heap translates those addresses into values.

In light of the modified computational-state construction, new assertions describing the heap are required. The notation for these assertions is shown in figure 2.4. In particular, the notation $e \mapsto e'$ is introduced to talk about the value (e') of an individual memory location (e) and the use of the star symbol ($p_1 * p_2$) denotes the disjointedness of memory locations p_1 and p_2.

Notation	**Name**	**Description**
emp	Empty Heap	The heap is empty.
$e \mapsto e'$	Singleton Heap	The heap contains one cell, at address e with contents e'.
$p_1 * p_2$	Separating Conjunction	The heap can be split into two disjoint parts such that p_1 holds for one part and p_2 holds for the other.
$p_1 -\!\!* \, p_2$	Separating Implication	If the heap is extended with a disjoint part in which p_1 holds, then p_2 holds for the extended heap.

Table 2.4: Separation Logic Heap Assertions

The great benefit of separating conjunction[3] and separating implication should not go unnoticed. These constructs facilitate the abstraction of otherwise complex issues about the size and order of the heap by giving us the advantage of local reasoning, which underlies the scalability of the logic. For example, the specification

$$\{256 \mapsto 10\} \ \texttt{C} \ \{256 \mapsto 50\}$$

implies not only that the program C expects to find the value 10 in heap cell 256, but also that this heap cell is the only addressable storage touched by the execution of C. In other words, separating conjunction and separating implication allows us to reason locally about programs while ignoring the rest of the storage. This scalability is not of much need for small programs but it is critical for more complex programs.

Additional axiom schemes for separating conjunction include: commutative and associative laws, the fact that **emp** is a neutral element, and various distributive laws. Some of these laws of particular interest for this research project are:

$$p_1 * p_2 \Leftrightarrow p_2 * p_1 \qquad \text{(Commutativity of *)}$$

$$(p_1 * p_2) * p_3 \Leftrightarrow p_1 * (p_2 * p_3) \qquad \text{(Associativity of *)}$$

$$p * \mathbf{emp} \Leftrightarrow p \qquad \text{(Idempotency of } \mathbf{emp}\text{)}$$

[3] An important consideration about notation is the difference between the syntax used for the singleton heap "$\mapsto$" and the symbol for logical implication "$\Rightarrow$". These symbols are not to be confused.

As for the inference rules, the statement-specific Hoare Logic rules remain sound, as well as the structural rules with the exception of the rule of constancy. This rule becomes unsound when switching from Hoare Logic to Separation Logic. Consider the example when the rule of constancy extends the specification of a valid Hoare triple for a program consisting of a simple assignment with an assertion that describes the value of a variable apparently not involved in the assignment as the evidence of such claim:

$$\frac{\{x \mapsto 0\} \;\; \texttt{[x]:=4} \;\; \{x \mapsto 4\}}{\{x \mapsto 0 \wedge y \mapsto 3\} \;\; \texttt{[x]:=4} \;\; \{x \mapsto 4 \wedge y \mapsto 3\}}$$

This example does not take aliasing into consideration (i.e. when $x = y$). When aliasing is possible due to the presence of memory pointers, the assignment of a new value to x will falsify the assertion in the postcondition that claims $y \mapsto 3$.

However, the ability to extend local specifications is still possible through the separating conjunction. In place of the rule of constancy, the *frame rule* which uses separating conjunction has been proposed:

$$\frac{\{P\} \;\; \texttt{C} \;\; \{Q\}}{\{P * S\} \;\; \texttt{C} \;\; \{Q * S\}}$$

By using the frame rule, it is possible to extend a local specification, involving only the variables and heap cells that are actually used by `C` (referred to as the *footprint* of `C`) by adding arbitrary assertions about variables and heap cells that are not modified or mutated by `C`. Thus, the frame rule is of great significance when dealing with procedure calls.

A procedure call could be considered as a subprogram that defines its own variables and heap cells; only these variables and heap cells, and those from the main program passed as parameters and defined as global variables, can be modified by a given procedure. Therefore, the local specification consists only of those variables modified by the procedure. When having a procedure call inside another procedure, the specification of the inner procedure can be extended by the variables and heap cells modified by the outer procedure and the specification for the inner procedure would remain valid.

Separation Logic axioms: In light of the new heap-manipulating statements introduced by Separation Logic, new logic constructs are to be introduced to describe the behaviour in terms of preconditions and postconditions of the mutation, deallocation, and allocation statements as listed in table 2.5. The axiom for mutation states that regardless of the value for the memory location to which e evaluates[4], the value will be updated with the result of evaluating the expression e'. The axiom for deallocation states that the result of disposing a memory location is an empty memory location. In case the disposed memory location was the only memory cell, this operation results in an empty heap. In case there are other memory locations, these are not modified by the dispose statement and are considered valid. Finally, the allocation axiom states that

[4] When it is desired to talk about a memory location which value is od no interest, the notation $e \mapsto -$ is used

the result of allocating a memory cell is the creation of a new memory location which value is the result of evaluating the expression $\bar{e}$.

Statement	Local rule	Global rule
Mutation	$\frac{}{\{e \mapsto -\}[e] := e'\{e \mapsto e'\}}$	$\frac{}{\{e \mapsto - * r\}[e] := e'\{e \mapsto e' * r\}}$
Deallocation	$\frac{}{\{e \mapsto -\}\texttt{dispose}\ e\{emp\}}$	$\frac{}{\{e \mapsto - * r\}\texttt{dispose}\ e\{r\}}$
Allocation	$\frac{}{\{emp\}v := \texttt{cons}(\bar{e})\{v \mapsto \bar{e}\}}$	$\frac{}{\{r\}v := \texttt{cons}(\bar{e})\{v \mapsto \bar{e} * r\}}$

Table 2.5: Axioms and Rules of Inference for Separation Logic

2.2.5 Calculus of Inductive Constructions (CIC)

In 1985, Thierry Coquand introduced a new formalism to construct proofs in natural deduction style called Calculus of Constructions. Based on the Curry-Howard isomorphism, stating the correspondence between propositions and types, the Calculus of Constructions provided a notion of a high-level functional programming language with complex polymorphism, well-suited for modular specification [15]. Based on Coquand's theory, the *Institut National de Recherche en Informatique et en Automatique*, developed a proof-assistant system called Coq[5]. The Coq system is designed for writing formal specifications, programs, and to verify that programs are correct with respect to their specification. The specification language called Gallina can represent programs as well as properties of these programs and proofs of these properties.

To prove a program in Coq, the properties of the target computer program are introduced as theorems which must be proven to hold valid given the formal description of the target system behaviour, a set of logical axioms, and a set of inference rules. Each step is checked by Coq to ensure the program does not have illogical leaps during the proof. When a property does not hold, Coq will encounter a proof requirement which cannot be discharged and which describes the circumstances of a bug. In other words, bugs in the system are found during the programming stage. Therefore, the resulting system is guaranteed to be bug-free in relation to the properties described.

There are currently many other proof assistants based on different logical frameworks, including other solutions based on the Calculus of Constructions, such as Isabelle, PVS, ACL2, and Twelf. Nevertheless,

[5] The underlying formal language of Coq is a Calculus of Constructions with Inductive Definitions, or Calculus of (Co)Inductive Constructions (CIC in short).

Coq presents some features that make it a more powerful tool to develop certified programs with respect to others [12]:

- Based on a Higher-Order Functional Programming Language - allows the programmer to make use of this familiarity with functional programming.

- Dependent Types - a language of dependent types allows to effectively capture any correctness property in a type.

- Easy to Check Kernel Proof Language - despite the complexity of some proof procedures, the final proof terms are expressed in a core language comparable with what is found in proposals for formal foundations for mathematics.

- Convenient Programmable Proof Automation - enables the user to build his own procedure for solving specific problems without allowing formulation of user-defined invalid proofs.

- Proof by Reflection - makes it easy to write programs that compute proofs by placing programs and proofs in the same syntactic class.

The rise of new technologies in the form of formal languages like Calculus of Constructions and proof assistants like Coq have given strength to formal verification of systems through theorem proving. In a strict sense, Coq is not an automated theorem prover since it requires interaction with the programmer to build proofs. Nevertheless, Coq makes use of a powerful set of theorem proving tactics and decision procedures that minimize the input required from the programmer. This tactic implements *backward reasoning*, proceeding from conclusion (goals) to premises (subgoals), replacing the goal with the generated subgoals. Therefore, when a tactic is applied, we say that the goal has been reduced to the subgoal. For example, the tactic *intro* reduces a goal of the form $T \to U$ to the subgoal U and gives a name (e.g. H_1) to the hypotheses T in the local context. We could then use the tactic *apply* H_1 to the new goal U to complete the original proof.

Since proving theorems in propositional logic is a very mechanical activity, it is possible to make use of more powerful tactics that automate the process as much as possible. For example, Coq's tactic *auto* implements a Prolog-like resolution procedure to solve the current goal. It first tries to solve the goal using the *assumption* tactic and then it reduces the goal to an atomic one by iteratively using *intro* and introducing the newly generated hypotheses as hints to the local context. Then it looks at the list of tactics associated with the goal and tries to iteratively *apply* one of them. This process is recursively applied to the generated subgoals. The tactic terminates when the goal has been solved or when it finds a proof requirement that cannot be discharged. [53].

2.2.6 Compcert

A clear example of the increased application of formal methods in the verification of software is Compcert [36], a formal verification of a realistic compiler. As a verified piece of software, Compcert comes with a machine-checked proof that guarantees the generated executable code behaves exactly as prescribed by the semantics of the source program.

The specification of CompCert's semantics was developed in Coq proof assistant; running a proof check determined that the desired properties for the compiler held from the provided specification. Once the proof was generated, executable code was extracted using Coq's extraction facility that generates executable Caml code from the Coq functional specifications [36].

The benefits of having taken the effort in producing a formal description for a compiler can be seen in the quality of the final product. A research group from the University of Utah created a randomized test-case generation tool and spent three years using it to find compiler bugs. The research group found that the middle- and back-end bugs[6] present in all other compilers, including gcc, are absent in CompCert [55]. The apparent unbreakability of CompCert supports a strong argument that developing compilers within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits.

2.2.7 CIC + Separation Logic

The use of Coq proof assistant is the difference between having a *machine-checkable* proof and having a *machine-checked* proof. The powerful expressiveness of Coq, thanks to its dependent type feature, allows us to have an implementation of Separation Logic so that we can manipulate the specifications as propositions, in a similar way we manipulate programs. Hence, after the specification of the program has been described in Separation Logic and the proof was built, we can execute the proof in Coq, allowing the computer to verify that we have not made any mistakes in building the proof.

Defining the syntax to express the specification of a program in Separation Logic is not difficult. The challenging part is constructing proofs in a proof assistant such as Coq because we are attempting to carry out linear-style reasoning in a proof assistant with a native logic that is not linear. For example, if A, B, C, and D are regular propositions (those used in Coq) then conjunction associativity, formally:

$$(A \wedge B \wedge C \wedge D) \Rightarrow (B \wedge (A \wedge D) \wedge C)$$

can be easily proved. On the other hand, if A, B, C, and D are Separation Logic assertions, proving the equivalent rule of associativity for the separating conjunction, formally:

[6] Note the emphasis on middle- and back-end bugs. This is because the parser and the runtime phases of Compcert were not validated in the same manner as the rest of the phases.

$$\forall m, (A * B * C * D)m \Rightarrow (B * (A * D) * C)m$$

is far more difficult due to the conditions for memory equality and disjointedness.

As an example, consider the work by Marti and Affeldt [40] in their formal verification of the heap manager of an existing embedded operating system. In their approach to verify a C implementation of a garbage collection, they made use of Separation Logic to express the assertions involved in the specification of the program behaviour. Nevertheless, they lacked the appropriate set of tactics to manipulate such assertions which forced them to unfold the definitions of assertions to allow the use of more conventional tactics. In other words, despite having a formalism to represent Separation Logic assertions, the proofs carried out were not at the same level of abstraction because of the lack of a mechanism to manipulate such formalism.

For this reason, an appropriate set of tactics are required to enable us to manipulate the specification at the right level of abstraction as to avoid complex, yet repetitive, reasoning. McCreight [41] provided a set of tactics for reasoning about Separation Logic assertions, including simplification, rearranging, splitting, matching and rewriting.

The simplification tactic reduces a Separation Logic assertion into a normal form in order to make further manipulation simpler. Some of the transformations include combining all instances of **true** and remove all instances of **emp** assertions. The rearranging tactic allows the commutativity and associativity properties of the separation conjunction to be applied in a concise and declarative fashion. The splitting tactic subdivides a Separation Logic proof by creating a new subgoal for each corresponding part of the hypothesis and goal. The matching tactic cancels out the matching part of the hypothesis with the goal. In case there is an assertion describing part of the heap in both, the hypothesis and the goal, then such an assertion is considered to be held in the specification and, therefore, eliminated. Finally, rewriting tactic provides support for logically equivalent assertions. By adding rewriting rules, the user can expand the simplification mechanism to work on their own assertions.

McCreight also provides a verified implementation of a Verification Condition Generator (VCG). The VCG is a weakest precondition generator derived from each individual statement along with the specification for the various ways to exit the statement. Verification requires that the user-specified precondition is at least as strong as the verification condition generated by the VCG. The soundness of the verification condition has been mechanically verified using Coq. Therefore, if the program under analysis is well-formed, then it is either possible to take another step or a valid termination state for the program has been reached.

Finally, a set of tactics to unfold the verification conditions and use Separation Logic to perform symbolic execution has been devised by McCreight. The automation of such tactics is what allows us to stay at the right level of abstraction to talk about the properties of FreeRTOS without having to deal with the details of the underlying logics.

The implementation of Separation Logic along with the verification condition generator and their respective appropriate tactics to manipulate the proof at the appropriate level of abstraction transform Coq into a proof assistant for Separation Logic. In other words, we now have an appropriate set of tools to provide a machine-checked proof of the correctness of safety properties of a real-time operating system. The next step is to carry out the proof of correctness of an actual system using the aforementioned tools. The next section introduces the details of the experiment carried out for FreeRTOS using the implementation of Separation Logic as a formal system to describe the behaviour of the system and whether or not the desired properties hold true for the given implementation.

Chapter 3

Experiment

A common approach to formally verify a piece of software using theorem-proving techniques is to start writing the implementation with a formal description of its specification in mind. This kind of solution consists of reasoning about simple constructs and proving them correct, iteratively building more complex pieces of software along with their corresponding proofs. One of the benefits of this approach is to be able to reason about the required proofs in the same way we reason about modularized software. On the down side, the programmer is required not only to reason about the required proofs, but also to provide an implementation of the system from scratch. This approach is referred to as *pre-development verification.*

Another approach to formal verification, referred to as *post-development verification* is to make use of existing software and apply post development techniques to specify the system's properties in a formal language. Once the specification is in a formal language, it is possible to apply natural deduction to provide the required proofs. As a potential side benefit for taking this approach, there exists the possibility of being able to undertake a similar exercise with other systems, opening the door of formal verification to existing pieces of software.

Since the focus of this research project is formal verification, it is, therefore, desired to reduce any complexity incurred by writing the source code for a real-time operating system. Hence, our approach is to use FreeRTOS — a professional-grade, existing implementation of a real-time operating system — and to write down the specification of its significant properties in a formal language to later be proved using Separation Logic embedded on Coq proof assistant.

To be able to manipulate FreeRTOS code, it is required to use an appropriate representation that strips away the details of the concrete syntax while keeping the semantics of the program. An Abstract Syntax Tree (AST) structure is the result of parsing the source-code concrete-syntax to determine its grammatical structure. Figure 3.1 shows the abstract syntax of a subset of C programming language supported by Compcert [6]. In general, the version of Compcer used in this experiment supports all of ANSI C, with a few exceptions that include:

- Unstructured `switch case` and `default` as in Duff's device,
- Unprototyped and variable-argument functions,

- Control operators `longjmp`, `setjmp`. `goto`, and labels,

- Storage class annotations `register`, `volatile`, `extern`, and `static` (although variables declared outside of functions are accepted as implicitly static), and

- Data structuring via `typedef`, `struct`, `union`, and `enum`.

In this experiment, Compcert's front-end is used to obtain the AST representation of FreeRTOS source code. Compcert is the implementation of a formally verified C compiler that comes with a machine-checked proof stating that the generated code behaves exactly as prescribed by the semantics of the source program. The architecture of Compcert is divided into three main parts: parsing, type checking and pre-simplification; compilation of Compcert C AST to assembly AST; and assembling and linking.

The first part transforms C concrete syntax into Compcert C AST. During this phase, some of the constructs not supported by Compcert C are expanded away. For example, block-scoped local variables are lifted up to function local scope; structs and unions passed by value are, instead, passed by reference; and assignments between structs and unions are eliminated. Some other unsupported constructs are rejected. In particular, assembly inline statements are not part of the subset of C supported by Compcert. This presented the first challenge when trying to obtain the AST representation of FreeRTOS. Therefore, some changes to the front-end of the compiler were needed in order to parse FreeRTOS due to the number of times assembly inlining is used in the source code. After extending the definition of supported statements to include assembly inlining, we were able to obtain the AST of FreeRTOS source code.

Once we obtained the FreeRTOS AST, we took advantage of a particular proof provided by Compcert: the semantics of the original program are preserved throughout all the intermediate languages/representations of the compiler [6]. Such a proof guarantees that the semantic analysis undertaken on a given intermediate representation will be preserved on all phases of the compiler, from the initial AST transformation to assembly language code generation. Given this property, working with *Cminor* — an intermediate language used in Compcert — (as opposed as working with C) is desirable for a number of reasons:

1. Most importantly, an axiomatic Separation Logic that has been proven sound with respect to its semantics has been implemented for Cminor. [1]

2. The language definition of Cminor is simpler than that of C while preserving the semantics of the original program written in C.

3. As the intermediate language in Compcert functioning as the interface between the front- and back-end of the compiler, Cminor could potentially serve as a common denominator between high-level languages by replacing the front-end of the compiler to translate a different language other that C.

Types:

$\mathit{signedness}$	$::= \texttt{Signed} \mid \texttt{Unsigned}$	
$\mathit{intsize}$	$::= \texttt{I8} \mid \texttt{I16} \mid \texttt{I32}$	
$\mathit{floatsize}$	$::= \texttt{F32} \mid \texttt{F64}$	
τ	$::= \texttt{Tint}(\mathit{intsize}, \mathit{signedness}) \mid \texttt{Tfloat}(\mathit{floatsize})$	
	$\mid \texttt{Tarray}(\tau, n) \mid \texttt{Tpointer}(\tau) \mid \texttt{Tvoid} \mid \texttt{Tfunction}(\tau^*, \tau)$	

Expressions annotated with types:

a	$::= b^\tau$	

Unannotated expressions:

b	$::= \mathit{id}$	variable identifier
	$\mid n \mid f$	integer or float constant
	$\mid \texttt{sizeof}(\tau)$	size of a type
	$\mid op_u\ a$	unary arithmetic operation
	$\mid a_1\ op_b\ a_2 \mid a_1\ op_r\ a_2$	binary arithmetic operation
	$\mid \texttt{*}a$	dereferencing
	$\mid a_1[a_2]$	array indexing
	$\mid \texttt{\&}a$	address of
	$\mid \texttt{++}a \mid \texttt{--}a \mid a\texttt{++} \mid a\texttt{--}$	pre/post increment/decrement
	$\mid (\tau)a$	cast
	$\mid a_1 = a_2$	assignment
	$\mid a_1\ op_b =^\tau\ a_2$	arithmetic with assignment
	$\mid a_1\ \texttt{\&\&}\ a_2 \mid a_1\ \texttt{\vert\vert}\ a_2$	sequential boolean operations
	$\mid a_1, a_2$	sequence of expressions
	$\mid a(a^*)$	function call
	$\mid a_1\ \texttt{?}\ a_2\ \texttt{:}\ a_3$	conditional expression
op_b	$::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%}$	arithmetic operators
	$\mid \texttt{<<} \mid \texttt{>>} \mid \texttt{\&} \mid \texttt{\vert} \mid \texttt{\^{}}$	bitwise operators
op_r	$::= \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{==} \mid \texttt{!=}$	relational operators
op_u	$::= \texttt{-} \mid \texttt{\~{}} \mid \texttt{!}$	unary operators

Statements:

s	$::= \texttt{skip}$	empty statement
	$\mid a;$	expression evaluation
	$\mid s_1; s_2$	sequence
	$\mid \texttt{if}(a)\ s_1\ \texttt{else}\ s_2$	conditional
	$\mid \texttt{while}(a)\ s$	“while” loop
	$\mid \texttt{do}\ s\ \texttt{while}(a)$	“do” loop
	$\mid \texttt{for}(a_1^?, a_2^?, a_3^?)\ s$	“for” loop
	$\mid \texttt{break}$	exit from the current loop
	$\mid \texttt{continue}$	next iteration of the current loop
	$\mid \texttt{return}\ a^?$	return from current function

Functions:

fn	$::= (\ldots \mathit{id}_i : \tau_i \ldots) : \tau$	declaration of type and parameters
	$\{\ \ldots \tau_j\ \mathit{id}_j; \ldots$	declaration of local variables
	$s\ \}$	function body

Figure 3.1: Abstract Syntax of the Supported Subset of C.

This characteristic allows the correctness of whole systems to be proven, even if the multiple components are written in different high-level languages.

With the representation of the computer program in the form of Cminor AST, Hoare triplets are constructed by describing the desired properties we want FreeRTOS to exhibit in terms of postcondition that must be true of the system given a stated precondition. The following section describes these properties.

3.1 Proving Significant Properties of FreeRTOS

Given the description of an RTOS in section 2.1 and based on the classification system by Lamport [35], the correctness properties of FreeRTOS are classified as follows:

1. **Safety properties**. A safety property asserts that nothing undesirable will happen during execution. In the context of operating systems, an example of this kind of property is mutual exclusion: given two different processes, it is desirable that they do not interfere with one another. This property is desired at different levels such as memory usage and processing time assignation, and can be applied to any shared resource, as is the case for memory.

2. **Liveness properties**. A liveness property asserts that something desirable will eventually happen. Due to the nature of real-time operating systems, liveness properties are strengthened to include a restricted time bound. In other words, a liveness property asserts that something desirable will happen within a stated time boundary. A common place for this kind of property in operating systems is found in the scheduling mechanism: given a task that has entered the ready queue, it is desired that the operating system will grant execution time to the task within a known amount of time.

The specific safety and liveness properties desired to hold in FreeRTOS are the focus of sections 3.1.1 and 3.1.2.

3.1.1 Safety Properties of FreeRTOS

As mentioned before, exclusive access is an important and common safety property in operating systems. In FreeRTOS this appears in the use of memory resources. To show that FreeRTOS guarantees exclusive access to memory it is first required to demonstrate a safety property of the memory manager that states that the memory allocation function will never allocate more memory than what is available. Later on, this property will be strengthened to demonstrate exclusive access of memory resources.

Allocator Bound Correctness FreeRTOS memory manager uses an array-like representation of the memory. The total size of the array (i.e. available memory) is set by the definition `configTOTAL_HEAP_SIZE`.

```
void *pvPortMalloc( size_t xWantedSize )
{
    void *pvReturn = ((void *)0);
    vTaskSuspendAll();
    {
        if( ( ( xNextFreeByte + xWantedSize ) < ( ( size_t ) 22000 ) ) &&
           ( ( xNextFreeByte + xWantedSize ) > xNextFreeByte ) )
        {
            pvReturn = &( xHeap.ucHeap[ xNextFreeByte ] );
            xNextFreeByte += xWantedSize;
        }
    }
    xTaskResumeAll();
    return pvReturn;
}
```

(a) C source code for heap_2.c

```
"pvPortMalloc"(xWantedSize) : int -> int
{
    var pvReturn, $2, $1;
    pvReturn = 0;
    "vTaskSuspendAll"() : void;
    if (int32["xNextFreeByte"] + xWantedSize <u 22000) {
        if (int32["xNextFreeByte"] + xWantedSize >u int32["xNextFreeByte"]) {
            $1 = 1;
        } else {
            $1 = 0;
        }
        $2 = $1;
    } else {
        $2 = 0;
    }
    if ($2) {
        pvReturn = "xHeap" + 1 * int32["xNextFreeByte"];
        int32["xNextFreeByte"] = int32["xNextFreeByte"] + xWantedSize;
    }
    "xTaskResumeAll"() : int;
    return pvReturn;
}
```

(b) Cminor source code

Figure 3.2: FreeRTOS Memory Allocator

Allocating memory in FreeRTOS consists of suspending the executing tasks to guarantee that nothing will interfere with the memory allocation. Once the tasks are suspended, the memory allocator checks that the size of the requested memory is valid (i.e. a positive value) and that there is enough free memory to serve the request. In case these conditions are met, the memory allocator returns a pointer to the first memory location that has been allocated and updates the value of the next free memory cell. The previously suspended tasks are resumed at the end of the procedure. The code for FreeRTOS memory allocator procedure and its transformation to Cminor using Compcert can be seen in figure 3.2.

Some of the differences between C and Cminor can be seen as result of this example transformation, including:

1. Types are instantiated to machine-specific formats and annotated. For example, every instance of `xNextFreeByte` in the C implementation is translated to `int32["xNextFreeByte"]` in Cminor.

2. The overloaded comparison operators are annotated with type discriminators. For example, the translation of the less-than operator in C (e.g. <) is translated to `<u` in Cminor.

3. Composed if-statements (i.e. those containing more than one condition) are expanded out into single if-statements with only one condition. To this extent, local variables are inserted to hold the logical result of each comparison. (e.g. variables `$1 and $2` are inserted in Cminor code).

One thing to notice is that FreeRTOS does not allow memory to be freed once it has been allocated. Therefore, the algorithm simply subdivides a single array into smaller blocks as requests for memory are made. `xNextFreeByte` points to the next free memory cell. Hence, at any point in time, all the memory with address `i < xNextFreeByte` are considered to be allocated while all those elements with index `i >= xNextFreeByte` are considered to be free and available for future memory allocation.

Knowing the details of the memory allocation algorithm (from now on known as *mallocBody*) permits the formal description of what it means for the memory management to not allocate more memory than available. More specifically, after the memory allocation function has been executed, the index of the next free memory location is less than the total size of the memory. In formal notation, this is initialized as:

$$mallocPost : \exists p, xNextFreeByte \mapsto p * (p < configTOTAL\_HEAP\_SIZE)$$

This predicate captures the essence of the desired final state (postcondition) of a memory allocator that does not exceed available memory. Nevertheless, it is necessary to add more detail about the rest of the variables involved in the process.[1] The complete postcondition states that the variables (`xWantedSize`, `xNextFreeByte`, and `xHeap`) and function calls (`vTaskSuspendAll`, and `xTaskResumeAll`) involved in the memory allocation procedure are valid memory locations and that the value held by the variable representing

[1] As it can be seen later, the rest of the conditions involved in the postcondition are simply carried through from the precondition.

the next free memory location is less than the total size of the memory. In formal terms:

$$\begin{aligned} \mathit{mallocPost} : \exists p,(& \mathit{xWantedSize} \mapsto a \\ & * \mathit{vTaskSuspendAll} \mapsto b \\ & * \mathit{xTasksResumeAll} \mapsto c \\ & * \mathit{xNextFreeByte} \mapsto p \\ & * \mathit{xHeap} \mapsto e \\ & * (p < \mathit{configTOTAL\_HEAP\_SIZE})). \end{aligned}$$

In order for the memory allocator to satisfy this postcondition, some precondition must also hold. To complete the Hoare triplet, the precondition is constructed. For this purpose, it would be ideal if there existed some kind of specification of FreeRTOS memory allocator, albeit informal, that could be translated into a formal description, but this is not the case, therefore a different approach is used.

Assuming that the implementation of FreeRTOS memory allocation function is correct, the required program state for the memory allocation function can be deduced from the function implementation. This assumption is based on the idea that FreeRTOS is a working operative system that has been implemented in different functional projects. Looking at the code shown in figure 3.2 we can see that `xNextFreeByte` is modified when the size of the memory requested is valid (i.e. a positive value) and there is enough free memory to serve the request. In terms of the actual implementation, these conditions are described as:

```
((xNextFreeByte + xWantedSize) < configTOTAL_HEAP_SIZE)
  && ((xNextFreeByte + xWantedSize) > xNextFreeByte)
```

These conditions takes care of the situations when the amount of memory requested is bigger than what it is available and when a process is requesting memory that has been previously allocated. Even though this condition seems to take care of all scenarios, it is still necessary to check for another in order to satisfy the desired postcondition: the case where `xNextFreeByte` is already bigger than `configTOTAL_HEAP_SIZE` even before the execution of the memory allocation function. In formal notation, this condition is defined as:

$$\mathit{mallocPre} : \mathit{xNextFreeByte} \mapsto d * (d < \mathit{configTOTAL\_HEAP\_SIZE})$$

An important difference between *mallocPre* and *mallocPost* is the use of the existential quantifier for the value pointed by `xNextFreeByte` in the latter one. This is due to the fact that before the execution of *mallocBody* we know (i.e. by inspection) the value of this variable. This is not the case for the value to hold by `xNextFreeByte` after the execution of *mallocBody* in which the final value depends on the execution path. Therefore, an existential quantifier is used as a placeholder for some value of which evidence can be shown after the execution of the memory allocator.

The precondition in its totality, including all variables involved in the code states that

- The memory location for the size of the memory requested (`xWantedSize)` is found in the heap.
- The function definition `vTaskSuspendAll` is located in memory.
- The function definition `xTaskResumeAll` is located in memory.
- The memory location for the next free memory location (`xNextFreeByte`)is found in the heap.
- The memory location pointing at the start of the heap (`xHeap`) is known and found in the heap.
- The value pointed to by `xNextFreeByte` is less than the total size of the heap.

In formal terms, this is expressed as follows:

$$\begin{aligned} \mathit{mallocPre} :& \mathit{xWantedSize} \mapsto a \\ & * \mathit{vTaskSuspendAll} \mapsto b \\ & * \mathit{xTasksResumeAll} \mapsto c \\ & * \mathit{xNextFreeByte} \mapsto d \\ & * \mathit{xHeap} \mapsto e \\ & * (d < \mathit{configTOTAL\_HEAP\_SIZE}). \end{aligned}$$

All the variables and function definitions called from within the body of the memory allocation procedure are needed in the precondition so that it is possible to step through the code implementation when building the proof. Failing to do so would result in a proof obligation that cannot be discharged due to the lack of enough information in the computational state. For example, in FreeRTOS memory allocator procedure shown in figure 3.2, the value of the variable `xNextFreeByte` is updated when a valid memory range is requested and there is enough free memory to serve the request. Failing to include the variable `xNextFreeByte` as a valid memory location in the precondition would result in falling to step through the assignment operation since the variable `xNextFreeByte` would not be known, which in turn would result in the impossibility to prove the required postcondition for FreeRTOS memory allocation procedure. Having all the required variables in the precondition *mallocPre* makes it possible to discharge all proof obligations as shown in Appendix A.

Once the precondition and postcondition required for the correct behaviour of the memory allocation function are defined, the next step is to define a theorem stating that, given the precondition *mallocPre* and the memory allocation function definition *mallocDef*, then the postcondition *mallocPost* holds after its execution. In other words, the Hoare triple {*mallocPre*} `mallocDef` {*mallocPost*} is now defined and can be instantiated in Coq as a theorem to be proven with the use of ancillary definitions as follows:

```
Lemma mallocOk : fdefOk P1 mallocTy 5%nat mallocPre mallocDef mallocPost.
```

The auxiliary function *fdefOk* proves that the specification of the Hoare triple holds. In a broad sense, it checks that given the precondition (*mallocPre*) and the current state of the program, following the logic of the program (`mallocDef`) it is possible to advance one step (i.e. the execution of such statement is valid). To accomplish this, three more arguments aside for the Hoare triple specification are needed. The code heap type (*P1*), the list of types of every argument received by the function definition (*mallocTy*) and the arity of the function definition (*5%nat*).

The code heap type provides a specification of the function definitions residing in memory. This is of special interest when dealing with function calls since it is necessary to check that the correct number of parameters are passed to each function, that the type of the parameters matches the type of the arguments expected by the function definition, and that calling a function will be placed in an expression where the return type of the function is expected.

In the memory allocation function, there are a couple of function calls, one to `vTaskSuspendAll` and one to `xTaskResumeAll`. This is reflected in the definition of the code heap type (`Definition` *P1*) as a function parameterized by the address (*c*) at which the function definition is to be found. Each address is composed by two elements: a memory block (*b*) and the block offset (*w*) to the desired location. The code heap type analyses the address passed as parameter and based on the offset it returns the specification corresponding to the function definition found in the given memory address. Since in this case the code heap type consists of only two function definitions, the analysis of what function is found at a given address simply requires checking whether the offset is zero (`match` *beq_nat* (*word_to_nat w*) 0), in which case the specification for `vTaskSuspendAll` (*suspendSpec*) is returned, otherwise, the specification for `xTaskResumeAll` (*resumeSpec*) is returned. Using Coq notation, this is defined as follows:

```
Definition P1 : cdhpty := fun c ⇒
  let (b, w) := c in
    match beq_nat (word_to_nat w) 0 with
      | true ⇒ Some suspendSpec
      | false ⇒ Some resumeSpec
    end.
```

Each function specification is defined by the type of its arguments, its arity, its precondition and its postcondition. For example, the specification for `vTaskSuspendAll` (*suspendSpec*) states that the function definition expects no parameters, and describes the precondition (*suspendPre*) and postcondition (*suspendPost*) as empty assertions. The implementation in Coq of this specification in described as follows:

> `Definition` *suspendTy* := (*nil* : *tslist*).
>
> `Definition` *suspendPre* := *emp*.
>
> `Definition` *suspendPost* (_ : *val*) := *emp*.
>
> `Definition` *suspendSpec* := *speci suspendTy* 0%nat *suspendPre suspendPost*.

where *suspendTy* is an empty list and together with the arity 0*%nat* specifies that the function `vTaskSuspendAll` receives no parameters.

At this point, all the definitions required to carrying out the proof of our theorem by using Coq proof assistant have been introduced. The details of such proof are omitted in this section and the interested reader is referred to Appendix A for a detailed structure of the proof stating that the memory manager won't allocate more memory than is available. Instead, a description of the structure of such proof is provided next.

The first step is the manipulation of the proof term as the setup to facilitate the handle of future proof terms. This includes the introduction of variable names for the existential quantifiers and the simplification of the proof environment, more specifically, the assertion describing the precondition.

At this point the first statements are stepped through using the tactic `vcSteps`. This tactic examines the current goal (i.e. the execution of a given Cminor statement) and checks for the required precondition (i.e. the state of the heap and the store) needed to successfully execute such statement. In case the precondition is met, the tactic will discharge the proof by the elimination of the statement from the next goal and the updating of the state according to the inference rule in Separation Logic that deals with the given Cminor statement. Due to the simplicity of the declaration and initialization of variables, this tactic steps through *mallocDef* until the call to `vTaskSuspendAll` is reached.

Stepping through a function call requires setting of the environment before the function call is executed. This setup process includes the manipulation of the code heap type to indicate where in the heap the function to be called can be found (i.e. `unfold P1`) and the specification of such function (`unfold suspendSpec` and `unfold suspendTy`). The tactic `callStep` is used to step through the function. The existence of a state that meets the requirements to function as the precondition for the next statement is then shown.

The succession of a nested `if` statement is found, followed by another `if` statement. The tactic `branchStep` is used three times to discharge each of the conditional statements. Since the inference rule dealing with `if` statement in Separation Logic has two premises, one for showing that the postcondition is true after the execution of the true branch when the condition is true and one for showing the postcondition is true after the execution of the false branch when the condition is false, this is reflected in the proof by the need to show that both premises are valid. Therefore, a total of six execution paths are required to be proven.

Finally, the same process done for the call to `vTaskSuspendAll` is repeated at this point to call the `xTaskResumeAll` function by using the tactic `callStep`.

Allocator Exclusive Allocation Another safety property of FreeRTOS is that memory is exclusively allocated: given a memory range `range(d,p)` starting at `d` and ending at `p` that has been assigned to a specific process, there will not be any other process to which a memory location within `range(d,p)` will be assigned. In other words, after the memory allocator has been executed, the newly allocated memory range will not have been previously assigned to any other process. In formal notation, this is defined as:

$$\exists p, \mathit{xNextFreeByte} \mapsto p * (\mathit{notPreviouslyAllocated}(d, p))$$

where d is the memory location pointed to by *xNextFreeByte* before the execution of the memory allocation procedure. This predicate captures the essence of the desired postcondition of a memory allocation function that enforces[2] exclusive allocation among processes. Again, it is necessary to add more details about the variables involved in the process. The complete postcondition for exclusive allocation states that the variables (`xWantedSize`, `xNextFreeByte`, and `xHeap`) and function calls (`vTaskSuspendAll`, and `xTaskResumeAll`) involved in the memory allocation procedure are valid memory locations and that the range of memory locations representing the allocated memory to serve the request has not been previously allocated. In formal terms, this is described as:

$$\begin{aligned}
\mathit{exclusionPost} : \exists p, (&\mathit{xWantedSize} \mapsto a \\
&* \mathit{vTaskSuspendAll} \mapsto b \\
&* \mathit{xTasksResumeAll} \mapsto c \\
&* \mathit{xNextFreeByte} \mapsto p \\
&* \mathit{xHeap} \mapsto e \\
&* (\mathit{notPreviouslyAllocated}(d, p))).
\end{aligned}$$

Using the same precondition as defined in predicate *mallocPre* and the memory allocation procedure definition (*mallocBody*) the Hoare triple is now complete and it is possible to define a theorem stating

[2] The memory allocation procedure cannot, by itself, guarantee exclusive access, only exclusive allocation. Even though guaranteeing exclusive access is a much stronger condition than ensuring exclusive allocation, some other conditions outside of the scope of the procedure must be met. For example, each process' code needs to satisfy this condition in the absence of hardware memory protection. Please refer to section 5.1.1 for a more detailed explanation.

that the postcondition *exclusionPost* holds true for the implementation *mallocBody* given the precondition *exclusionPre*. The ancillary function *fdefOk*, the code heap type (*P1*), the type (*mallocTy*) and number of arguments (5%nat) work as previously described in `Lemma` *mallocOk*. In formal terms, this is given as:

`Lemma` *exclusionOk* : *fdefOk P1 mallocTy* 5%nat *exclusionPre mallocDef exclusionPost*.

A proof of this theorem has a very similar structure to the proof previously described. Therefore, the details are left out of this section. The interested reader is referred to Appendix B for a complete explanation of a machine-checked proof of FreeRTOS exclusive allocation.

3.1.2 Liveness Properties of FreeRTOS

Liveness properties are commonly required of the operating system's scheduler [17] and depending on the scheduling mechanism, these kind of properties can take different shapes. In a round-robin scheduler, the correct implementation of a scheduling mechanism will guarantee that once a task has been moved from the *waiting* queue to the *ready* queue, the operating system will grant it scheduling time. For a priority-based scheduler, this property can be a little different.

The design of a priority-based scheduling mechanism does not guarantee that all tasks will be given processor time. There are certainly a number of scenarios in which a task could never be granted processing time. For example, consider the case of an implementation in which the processing time is assigned first to highest priority tasks while lower priority tasks are executed only when the higher priority tasks have been completed. Moreover, consider that once the task priority has been assigned, it cannot be changed. When a running $task_A$ with $priority = MAX\_PRIORITY$ is designed to run without sleeping, other tasks will never be scheduled. Although the scheduler might execute the `vTaskSwitchContext` procedure, any other task that is created (i.e. $task_B$) with $priority < MAX\_PRIORITY$ will never be granted processing time since the scheduler will always choose the same $task_A$ due to its highest priority.

The implementation of FreeRTOS' scheduling mechanism is priority-based and it follows the policies described in the previous example. Each task is created with a fixed priority ranging from 0 to *(configMAX_PRIORITIES - 1)*. Low priority numbers denote low priority tasks. The FreeRTOS scheduler ensures that tasks in the *ready* or *running* state will always be given processor time in preference to tasks of a lower priority that are also in the *ready* state. In other words, the task placed into the *running* state is always the highest priority task that is able to run. Moreover, once a task priority has been assigned it cannot be changed.

Since the implementation of FreeRTOS cannot guarantee that every single task will be granted processing time, the liveness property of the scheduler takes an unusual form: the procedure in charge of selecting the next executing tasks will terminate in a known amount of time with the highest priority task selected for

```
unsigned int factorial(int n)
{
  if(n==0) {
    return 1;
  } else {
    return n * factorial(n-1);
  }
}
```

Figure 3.3: Recursive Factorial

execution. In other words, FreeRTOS' scheduler will guarantee that the `vTaskSwitchContext` procedure will always terminate its execution, and when it does, the task selected for execution will be the one with the highest priority among the *ready* tasks. This property differs from the previous safety properties in that it not only requires the `vTaskSwitchContext` procedure to be safe (*partial correctness*) but it also requires the procedure to guarantee its termination (*total correctness*).

In the partial correctness setting, proofs come with a disclaimer: **only if** the program terminates its execution does a proof of $\{P\}Q\{R\}$ tell us anything about that execution. Partial correctness does not tell us anything if Q loops forever. Total correctness extends partial correctness by also requiring that Q terminates.

Proving total correctness requires a special analysis of those constructs that could lead to infinite iterations. Examples of this kind of constructs in C programs are the following: recursion and infinite (non guarded) loops.

Infinite recursion is when a program makes a non-terminating number of function calls. A very common example of recursion is the program that calculates the factorial of a given number shown in figure 3.3. In this program, the factorial function makes a recursive call to itself by passing a different argument — the predecessor of the original argument. Thanks to the base case where there is no recursive calls (when `n==0`) and the decreasing nature of the arguments, this function will eventually terminate.

Nevertheless, the potential of a malformed recursive procedure can lead to an infinite number of calls, causing the program to never terminate. For this example, consider the modified factorial program shown in figure 3.4. Even though there exists a base case and the argument is decreasing, there could be an execution trace in which the base case is never met, in particular, any call to `factorial` of an odd number would cause the program to never terminate.

The other construct that can lead to nonterminating programs is non-guarded loops. Contrary to guarded loops when the body of the loop is executed a fixed number of times, with non-guarded loops it is impossible to predict the number of times the body of the loop will be executed based on code examination. A non-guarded loop usually tests for the occurrence of a particular event, not a count of the number of repetitions. An example of this kind of loop is commonly found when dealing with streams and files. Figure 3.5 shows a

```
unsigned int factorial(int n)
{
  if(n==0) {
    return 1;
  } else {
    return n * (n-1) * factorial(n-2);
  }
}
```

Figure 3.4: Infinite Recursive Factorial

```
void printFileContent(FILE* file_handler){
    while(!feof(file_handler)) {
      fprintf(stdout,"%s\n",fgets(file_handler));
    }
}
```

Figure 3.5: Non-guarded Loop — The loop condition test for the happening of an event

loop that tests for a specific event (i.e. the end of the file) instead of counting the number of repetitions.

A special case of non-guarded loops is when the body of the loop (or external event) does not change the condition of the `if` statement, causing the loop to iterate infinitely. An example of this type of loop is shown in figure 3.6.

An analysis of FreeRTOS source code shows that recursive functions do not exist at all — not a surprising fact of a real time operating system, where explicit deadlines are required to be met. In other words, not only there is not a function recursively calling itself, but also function calls do not form a loop that could potentially cause the program to not terminate.

On the other hand, it is hard to devise an operating system that makes no use of iterations — the other possible cause of non-terminating programs. FreeRTOS is no exception to this situation. Iterative operations are easily found in FreeRTOS, especially when dealing with the data structures that contain the different tasks representing the processes. To show that FreeRTOS meets the execution deadlines, it is of prime interest to show that the iterative operations will eventually terminate.

The procedure in charge of going through the list of tasks and selecting the one with the highest priority as the next running task is called `vTaskSwitchContext`. Its implementation in C and Cminor are shown in figure 3.7.

```
unsigned int loop() {
  while(true){
    printf("Infinite␣loop");
  }
}
```

Figure 3.6: Non-guarded Loop — The loop condition never changes during the execution of the program

```
void vTaskSwitchContext( void )
{
    if( uxSchedulerSuspended != ( unsigned long ) ( 0 ) )
    {
        xMissedYield = ( 1 );
        return;
    }
    while(((&(pxReadyTasksLists[uxTopReadyPriority]))->uxNumberOfItems
           == (unsigned long)0))
    {
        uxTopReadyPriority = uxTopReadyPriority - 1;
    }
    {
        xList * const pxConstList = &( pxReadyTasksLists[ uxTopReadyPriority ] );
        ( pxConstList )->pxIndex = ( pxConstList )->pxIndex->pxNext;
        if( ( pxConstList )->pxIndex == ( xListItem * )
           &( ( pxConstList )->xListEnd ) ) {
                ( pxConstList )->pxIndex = ( pxConstList )->pxIndex->pxNext;
        }
        pxCurrentTCB = ( pxConstList )->pxIndex->pvOwner;
    };
}
```

(a) C source code tasks.c

```
"vTaskSwitchContext"() : void
{
    var pxConstList, $16, $15, $14, $13, $12, $11, $10, $9, $8, $7, $6, $5, $4,
        $3, $2, $1;
    $1 = int32["uxSchedulerSuspended"];
    if ($1 !=u 0) {
        int32["xMissedYield"] = 1;
        return;
    }
    {{
        loop {
            if (! 1) {
                exit 1;
            }
            {{
                $2 = int32["uxTopReadyPriority"];
                $3 = int32["pxReadyTasksLists" + 20 * $2 + 0];
                if (! ($3 ==u 0)) {
                    exit 2;
                }
                $4 = int32["uxTopReadyPriority"];
                int32["uxTopReadyPriority"] = $4 - 1;
            }}
        }
    }}
    $5 = int32["uxTopReadyPriority"];
    pxConstList = "pxReadyTasksLists" + 20 * $5;
    $6 = pointer[int32[pxConstList + 4] + 4];
    int32[pxConstList + 4] = $6;
    if (int32[pxConstList + 4] == pxConstList + 8) {
        $7 = pointer[int32[pxConstList + 4] + 4];
        int32[pxConstList + 4] = $7;
    }
    $8 = pointer[int32[pxConstList + 4] + 1]];
    pointer["pxCurrentTCB"] = $8;
}
```

(b) Cminor source code

Figure 3.7: FreeRTOS Task Switch procedure

```
local2 = [uxTopReady];
local3 = [pxReadyTaskList] + local2;
while(local3 == 0) {
    local4 = [uxTopReady];
    [uxTopReady] = local4 - 1;
    local2 = [uxTopReady];
    local3 = [pxReadyTaskList] + local2;
}
```

Figure 3.8: FreeRTOS Task Selector

This procedure can be divided into three main parts for analysis. The first part checks whether the scheduler is in suspended mode, in which case the global flag `xMissedYield` will be updated and the rest of the `vTaskSwitchContext` procedure won't be executed. The second part is a simple loop that walks through the list of *ready* tasks, starting at the index of highest priority, in search for the first non-empty position indicating the existence of a task with priority `uxTopReadyPriority`, the index of the list decreases by a single unit in case no such task exists in search for the next highest priority task. In the third part inside a block of code, the data structure containing the task is updated to account for the removal of the task, and the process control block is updated — which is the data structure in the operating system kernel containing the information to manage the process.

As in the previous example shown in figure 3.2, some of the differences between C and Cminor include the instantiation and annotation of machine-specific formats, as well as the annotation of overloaded comparison operators. In figure 3.7 we see even more of the differences between C and Cminor, including:

1. The notation for access to `struct` elements in C is eliminated and replaced by direct pointer arithmetic. For example, the index `i` of the list of *ready* tasks is accessed by the expression (`pxReadyTaskList + 20 * i`) where `20` is the size of the structure representing each element in the list.

2. The `while` loop is translated into a simpler Cminor `loop` construct which ending condition is located in the block of code inside the body of the loop. For this purpose, the Cminor `exit` ⟨*exp*⟩ construct is introduced. The evaluation of the expression of the `exit` statement indicates the number of levels (i.e. nested loops, nested blocks of code, and/or a combination of both) to jump out.

The only construct in Cminor that could lead to non-termination is `loop`. Any `loop` element is the result of a transformation from any C loop (namely `for`, `while`, and `do-while`) to a simpler form that loops infinitely until an explicit condition is met.

The implementation of Separation Logic and Cminor differ in the way this loop transformation gets done. Because of this, the code shown in figure 3.7 had to be modified to accommodate for the differences between implementations. The resulting while loop syntax, from now on referred to as *whileBody*, is shown in figure 3.8.

Following the Hoare Logic *while rule* from table 2.2, it is first needed to state the loop invariant P — the condition that must be satisfied before, during, and after the loop (regardless of whether the body of the loop gets executed or not). In this case, the condition for the loop invariant is that the element with the highest priority (represented by the variable `local3` in *whileBody*) is bigger than, or equal to 0. In formal notation, this is represented as:

$$\exists p, local3 \mapsto p * (p \geq 0)$$

Since the while loop is guarded by the condition `local3 == 0`, this condition becomes our guard B — the condition that is true before executing the body of the while statement and becomes false after its completion. $P \wedge \neg B$ is the postcondition, this is written in formal terms as:

$$taskSwitchPost : \exists p, local3 \mapsto p * (p > 0)$$

It is now possible to carry out a partial proof of correctness, stating that the postcondition will hold for *whileBody* given the precondition **only if** the program terminates. Since a proof of total correctness is required, it is necessary to extend the pre- and postconditions with a *loop variant*.

A loop variant is an integer expression whose value is always positive and can be shown to decrease every time that body of the while-statement is executed. If it is possible to find a loop variant, it follows that the while statement must terminate; because, the variant can only be decremented a finite number of times before it becomes 0 [30].

It is possible to codify this intuition in the following rule for total correctness, which replaces the rule for the while statement.

$$\frac{\{P \wedge B \wedge 0 \leq [E] = [E_0]\}\ \texttt{C}\ \{P \wedge 0 \leq [E] < [E_0]\}}{\{P \wedge 0 \leq E\}\ \texttt{while B do C end}\ \{P \wedge \neg B\}}$$

For the case of *whileBody*, the expression that meets the requirements to function as the loop variant is `uxTopReady = uxTopReady - 1`. The condition that `uxTopReady` $\geq$ `0` is true before the execution of the loop (due to the precondition that requires that the list of tasks is not empty[3]). Moreover, the condition that the expression decreases on each iteration (`uxTopReady = uxTopReady - 1`) and stays always positive during the execution is also met by such expression.

[3] This condition is met by having a lowest priority *idle* task always present in the *ready* queue as described in section 2.1.1.

The complete precondition including the loop invariant, the loop variant, and all other variables involved in the execution of the while loop is shown next:

$$\begin{aligned} taskSwitchPre : & uxTopReady \mapsto a \\ & * pxReadyTaskList \mapsto b \\ & * local3 \mapsto p \\ & * (listNotEmpty(b)) \\ & * (p \geq 0) && \text{(invariant)} \\ & * (b \geq 0). && \text{(variant)} \end{aligned}$$

It is possible now to prove the theorem that states that the procedure in charge of selecting the next executing task is not only partially correct, but it will eventually terminate. The details of the proof are omitted in this section. The interested reader is referred to Appendix C for a detailed description of the theorem as well as its proof.

The last proof gives us the confidence that, if the preconditions are met, the taskSwitch procedure will never run infinitely.

Task Switch Bounded The property of termination might be sufficient for most operating systems, but in the case of a real-time operating system, it might not be the case. The description of a real-time operating system in section 2.1 requires not only the certainty that the system will eventually terminate, but the bound time is also of interest. For this reason, it is desirable to extend our theorem for total correctness with an explicit deadline representing the upper limit bound (worst-case scenario) of the time the program will take in completing the task.

The proposed solution consists of a cost-dynamics analysis. Due to the use of axiomatic semantics analysis undertaken by the experiment, the ability to keep track of individual steps to complete an operation is lost. In contrast to operational semantics, the number of steps required to evaluate to an expression cannot directly be determined. Instead, the evaluation relation must be augmented with a cost measure, resulting in a cost dynamics. [28]

The proposed cost measure consists of the sum of the CPU cycles of the assembly instructions into which the program compiles. To this end, Compcert's backend is used to transform the Cminor program to assembly code. Note that Compcert preserves the behaviour of the original program throughout all the intermediate languages/representations; so, the analysis undertaken for the assembly code is guaranteed to be also valid for any other representation, including Cminor and the machine executable code. The assembly code for the piece of code corresponding to the `vTaskSwitchContext` procedure is shown in figure 3.9.

```
vTaskSwitchContext:
    mov r12, sp
    sub sp, sp, #16
    str r12, [sp, #12]
    str lr, [sp, #8]
    str r4, [sp, #0]
    ldr r0, .L258 @ uxSchedulerSuspended
    ldr r0, [r0, #0]
    cmp r0, #0
    beq .L259
    ldr r3, .L260 @ xMissedYield
    mov r1, #1
    str r1, [r3, #0]
    b   .L261
.L259:
    ldr r0, .L262 @ uxTopReadyPriority
    ldr r0, [r0, #0]
    ldr r2, .L263 @ pxReadyTasksLists
    mov r1, r0, lsl #4
    add r3, r1, r0, lsl #2
    add r0, r2, r3
    ldr r0, [r0, #0]
    cmp r0, #0
    bne .L264
    ldr r0, .L262 @ uxTopReadyPriority
    ldr r0, [r0, #0]
    ldr r1, .L262 @ uxTopReadyPriority
    sub r3, r0, #1
    str r3, [r1, #0]
    b   .L259
.L264:
    ldr r0, .L262 @ uxTopReadyPriority
    ldr r1, [r0, #0]
    ldr r3, .L263 @ pxReadyTasksLists
    mov r2, r1, lsl #4
    add r2, r2, r1, lsl #2
    add r4, r3, r2
    ldr r1, [r4, #4]
    add r3, r1, #4
    ldr r0, [r3, #0]
    str r0, [r4, #4]
    ldr r2, [r4, #4]
    add r3, r4, #8
    cmp r2, r3
    bne .L265
    add r0, r2, #4
    ldr r1, [r0, #0]
    str r1, [r4, #4]
.L265:
    ldr r0, [r4, #4]
    add r0, r0, #12
    ldr r2, [r0, #0]
    ldr r0, .L266 @ pxCurrentTCB
    str r2, [r0, #0]
.L261:
    ldr r4, [sp, #0]
    ldr lr, [sp, #8]
    ldr sp, [sp, #12]
    mov pc, lr
.L263:  .word   pxReadyTasksLists
.L260:  .word   xMissedYield
.L262:  .word   uxTopReadyPriority
.L266:  .word   pxCurrentTCB
.L258:  .word   uxSchedulerSuspended
```

Figure 3.9: Assembly Code for `vTaskSwitchContext` Procedure

Since the assembly code is produced by Compcert, all there is left to complete a cost analysis is to add an extension to Compcert that computes the sum of the cost of each individual instruction in the assembly code based on the specification for the instruction set architecture on which the code will be executed. In the particular case of this experiment, the selected target is an ARM7 processor with an ARM7TDMI instruction set, for which ARM provides extensive documentation, including the execution cost [38].

Adding a cost-analysis extension to a compiler is not a difficult task, but in the case of Compcert, it requires special care. Changes to the compiler requires not only to be careful to not break Compcert's code, but they also must preserve its proof of correctness. This situation is inherent of most programs developed in Coq, where a program is not merely a sequence of instructions but it may come with a proof of correctness. Adding an extension to Compcert turned out to be an interesting software engineering experiment in itself. More details on the design decisions and implications of this extension are presented in section 4.2.5.

Running the `vTaskSwitchContext` procedure through Compcert and the cost analysis program results in a calculation of 4357 cycles. To make this result concrete, consider the evaluation board AT91SAM7X256 for which a realistic demo has been developed using FreeRTOS. The ARM processor in this board operates at 55MHz. Therefore, the `vTaskSwitchProcedure` will finish its execution in 9μs per priority level.

While this calculation is considered accurate, it is important to specify its limitations to know what this result is really telling us. There are three limitations that undermine the accuracy of the cost of execution: simplification across execution paths, the lack of consideration for the size of the data structures, and its naiveness with regard to hardware optimization. Nevertheless, the time boundary is always overestimated. In other words, the results are still considered valid, in that there is a fixed task switch deadline.

The first limitation is simplification across execution paths. For instance, consider the case of an if-else-statement. This conditional construct will only execute one of the two possible paths depending on whether the condition is true or false. One cost estimation would be the maximum cost of both paths treated separately. A less accurate cost analysis, used in this experiment, adds the cost of both execution paths, even though in reality this will never happen. As stated before, this causes an overestimation of the cost but the results would be valid for those applications that can afford a response time of 9μs per priority for the `vTaskSwitchContext` procedure. In case a better response time is required there is room for improvement.

The second limitation is the lack of consideration for the size of data structures. As an example consider the `while`-loop in the `vTaskSwitchContext` procedure. This loop traverses the list of ready tasks looking for the next task to be executed. Therefore, the operations inside the loop will be executed as many times as the dimension of the data structure (in the worst-case scenario, as many times as `configMAX_PRIORITIES`). The solution presented in this experiment does not consider this situation and makes it necessary to perform this calculation manually. In other words, the result of 9μs only considers the execution of a single iteration. To accommodate the other possible (and highly likely) iterations, we need to multiply the result by a factor of configMAX_PRIORITIES.

Finally, the calculation is naive with regard to hardware optimizations. In particular, the cost-analysis does not consider the 3-stage pipeline used to increase the flow of instructions to the processor; instead, the program considers every instruction to be executed sequentially and individually.

In this experiment, evidence of the proof of safety properties of FreeRTOS has been provided, with special focus on the memory manager module. Also, the task switch mechanism is proven to eventually terminate and an estimate of the execution cost for such operation has been established, providing a strong liveness property. Clearly, the representative significant properties of a real-time operating system can be verified using Coq, augmented with Separation Logic and a certified compiler.

The evaluation of this experiment, including the development tools and a comparison of this verification methodology to other related work is presented in the next chapter.

Chapter 4

Evaluation

After showing that the Coq proof assistant extended with Separation Logic is an appropriate set of tools for the verification of a real-time operating system, it is now important to discuss the substantive arguments against formal verification; the complexity that formal verification represents is among the most recurrent ones [27].

Defining a metric to judge the complexity of an intellectual task is not easy due to the number of factors that come into play, many of them of subjective nature. One measure is size of outputs; we provide a software engineering-based metric intended to describe the effort required in proving the safety and deterministic properties of FreeRTOS. This metric is based on the lines of code (LOC) that each one of the proofs required.

The LOC metric is considered proportional to effort in the sense that it takes time to produce each line of code without taking into account blank lines or comments; the more lines of code, the longer it takes to produce. The time it takes to produce a single line of code depends on many factors, some harder to measure than others. One of the measurable factors is the number of words that each line contains. Following the same correlation as the LOC, the higher the word count in a single line, the longer it takes to produce.

A distinction is made between the work required to setup the problem definition and carrying out the interactive proof. Definitions consist of code required to specify the problems in terms of variables and coding notations, as well as to import the required libraries for Separation Logic. The definitions for *time of execution* also account for the Coq function definitions to carry out the cycles calculation. For the discharge of a proof, each LOC corresponds to a tactic call while each word corresponds to either the name of the tactic, environment terms (i.e. variables and hypotheses names), or newly introduced terms.

The complexity analysis in terms of LOC and word count for each of the proofs of FreeRTOS' memory manager and scheduler modules are presented in table 4.1. The order in which the verified properties are listed follows the order in which the proof obligations were produced. In general, with an average of less than 60 LOC, the definition for the memory safety and scheduler liveness properties were easy to describe but discharging the proof obligations presented a bigger challenge (an average of over 100 LOC). The low ratio of words/LOC (averaging less than 4 words/LOC) is due mainly to the low level of automation of the tactics required for the proof. Since each tactic performs a simple operation, the amount of information required in terms of received arguments is low. In contrast, the cost-analysis tool required to calculate the execution time of a given procedure assembly code required more effort to define (more than twice the number of LOC when compared to the safety and liveness properties) but proving the required proof obligations was relatively easier (less than have the number of LOC compared to the memory safety properties). The high ratio of words/LOC is mainly due to the high level of automation of the tactics used in the required proofs.

Property	**Proof Obligations**	**Definitions (LOC)**	**Proof (LOC)**	**Proof (words)**
Memory manager won't allocate more memory than available	1	55	120	402
Memory will be exclusively allocated	1	56	133	421
Scheduler will eventually select highest priority task	2	57	81	259
The time of execution for the `vTaskSwitchContext` procedure is known and bounded on the top	7	134	67	351

Table 4.1: Proof Metrics

There is a steep learning curve for the tools involved in the development of this work. Nevertheless, the effort in mastering both tools proved to be worthwhile as the experiment progressed. The difference in the time it took to develop the first proof and the subsequent proofs was substantial. Once the significant properties to be proved were defined, I proceeded to prove the memory safety property stating that the allocation procedure won't allocate more memory than available. This exercise took about one third of the total time spent in the whole project. After the memory safety proofs were completed, the efforts were focused on proving the liveness properties of FreeRTOS scheduler. Considering the higher number of proof obligations in the much shorter amount of time gives us an appreciation on the benefit of learning the tools.

The evaluation of the experience in developing the proof of correctness from a software verification point of view is presented in the following section.

4.1 Software Verification Standpoint

The fundamental reason to carry out this experiment was the exercise of formal verification techniques to verify the correct implementation of a software system. With this in mind, the evaluation of the verification approach is evaluated based on the criteria suggested by Huth and Ryan [30] along with the comparison of its advantages and disadvantages with other verification approaches.

4.1.1 Proof-based vs. Model-based

As we mentioned before, both model checking and theorem proving have been successfully applied as verification tools [7]. Both approaches embed a formal language strong enough to proof safety and liveness properties, resulting in the continual development and improvement of tools for both approaches.

The main advantage of using proof-based techniques is its feasibility to be implemented in large-scale systems. A real-time operating system is a complex piece of software consisting of different modules making use of complex data structures. The state explosion problem of model checking makes it unfeasible to describe a real-time operating system in terms of low-level state changes.

A common disadvantage of a proof-based approach is the lower degree of automation compared to model checking techniques. Model checking makes use of highly automated tools that simplify checking whether a property holds true of a system. The proof-based approach in this experiment had to be guided by a developer. Whereas this seems to be a big disadvantage, the next section shows how automation finds its place in proof-based methods.

4.1.2 Degree of automation

Since the beginning of formal methods, there has been an increase in the number of tools designed to aid formal verification. For a proof-based approach, the tools can be classified in two main categories:

1. Automated theorem provers such as ACL2 and PVS.

2. Proof assistants such as Coq and Isabelle.

The idea of feeding an automated theorem prover the formula describing the property of the system requiring verification and getting the proof as a result without the need of human intervention seems highly attractive, but this does not come for free.

One disadvantage of automated theorem provers is that not every formula is possible to prove and deciding whether a formula is provable or not is not possible. In these cases, an invalid formula could be entered and the system would not be able to recognize it as invalid, causing the automated theorem prover to fail to terminate while searching for a proof.

For this reason, using a proof assistant is a better approach. The use of proof assistants requires human guidance to provide a proof; therefore, it is possible to leverage the expertise and insight of the programmer to guide the computer through the logical steps that a proof requires.

Moreover, even in proof assistants there is a place to increase the level of automation in the form of tactics. Defining new tactics allows the programmer to scale up the efforts to more complex definitions and more interesting properties without becoming overwhelmed by repetitive or low-level details. Consider the example of the omega tactic — a decision procedure for Presburger Arithmetic. If the goal is a universally-quantified formula made out of numeric constants, addition, subtraction, multiplication, equality/inequality and first-order logical connectives, then invoking omega will either solve the goal or indicate that the goal is actually false [46].

The degree of automation in this experiment falls somewhere in the middle of the range of fully automated and fully manual. Much of this is thanks to the implementation of Separation Logic and its appropriate set of tactics as well as the extensive standard library provided by Coq. The combination of these tools eliminated the low-level details of the logic of CIC and the embedding of Separation Logic into Coq.

It certainly could be possible to increase the level of automation in the experiment by designing a set of tactics that deal with the common scenarios present in the proofs, but in doing so we run the risk of making the solution highly domain-specific. A balance between the two extreme cases gives us the benefit of both approaches.

4.1.3 Full vs. property-based verification

In principle, full verification involves proving that a complete design is correct and complete with respect to its specification. Hence, proofs of correctness are carried out to verify the functional behaviour of each component and of the overall system. It may be the case, however, that not all functional properties have the same degree of importance. A possible development choice is to restrict the application of formal methods to only the verification of selected properties.

This experiment presents the proof of safety and liveness properties of a real-time operating system. This property-based verification approach has the benefit of modularizing the solution: making the problem easier to solve. For example, while verifying the property that FreeRTOS enforces exclusive memory allocation among processes, it was only necessary to focus on the memory manager module of the operating system while ignoring the detail of other modules.

Another advantage of property-based verification is the ability to apply formal methods to other subsystems that might not be considered critical as a whole, but that contain critical modules within them. There is a common misconception that formal verification is only required of critical systems [27] but with a property-based approach, formality can be brought to noncritical systems by applying them only to subsystems or components in the case the effort of verifying the full system is considered prohibitive. Components that are not safety critical, on the other hand, can be verified with more traditional methods or just tested.

Finally, a property-based approach can be extended to include all properties of the system, making it as complete as a full-verification. In other words, property-based verification provides the advantages of a modularized development, allowing for separation of concerns and collaborative and incremental development.

If done extensively, property-based verification yields full-based verification.

During the last three decades, there has been substantial work on full implementations of a verified operating system kernel. Earlier efforts are exemplified by the UCLA Security Kernel [54] and the work by Bevier [5]. The use of limited and less-suitable logics such as Boore-More and predicate calculus makes both efforts remarkable. A question to ask is: how much can we trust the software tools that both projects used? Arguably, the formal verifier could have produced the wrong results, for instance, due to a software bug affecting it or due to a bug in the compiler used for object code generation.

A significant advantage of this experiment's approach is the use of a trusted set of tools that aided its development. Compcert and Separation Logic are the cornerstones of this verification exercise. The use of both tools not only simplified the development of FreeRTOS verification; but, they guarantee their own correct behaviour. Building on top of their foundation increases the confidence of the verification framework proposed by this work.

More recent related work provides a higher level of confidence about the tools used in the complete implementation of a verified operating system; such is the case of seL4 [33] — a formal, machine-checked verification of a microkernel from an abstract specification down to its C implementation. This work is impressive in that all the proofs were done inside Isabelle/HOL — a high-assurance mechanized proof assistant comparable to Coq.

The work in this thesis differs from seL4 is in the set of tools that were available for their respective proof assistants. Using Coq, we took advantage of the substantial mindshare that has accumulated throughout the years around it. The implementation of Compcert and Separation Logic are the best two examples but documentation also helped. Lacking similar tools for Isabelle/HOL (in part due to the starting time of the project) forced the seL4 group to significantly increase the effort in producing the proofs of correctness. In essence, seL4 refinement proof mimics Compcert's main theorem that the behaviour of the program is preserved throughout the many compilation phases. These extra proof obligations certainly increased the time it took to prove seL4 correctness [33].

4.1.4 Intended domain of application

Application of formal methods in industry has increased over the years [7]. Within the wide range of industrial fields on which formal methods have been applied, safety-critical systems are particularly noteworthy because formal methods can be used to support their certification. Certification is typically a necessary step that must be achieved before a safety-critical system can be deployed and used [8]. In the certification context, a regulatory authority establishes a set of requirements and guidelines that the product must satisfy in a published standard. In some of these standards, formal methods are recommended, or at least admitted as an alternative with respect to more traditional techniques such as testing, to carry out the system development and certification.

The developers of FreeRTOS provide a certified version of a real-time operating system known as SafeRTOS. The certification process for this operating system is based on structural coverage of its source code to test for each condition of each decision within the program. Even when such coverage testing is sufficient evidence for complying with standards under which SafeRTOS is certified (DO-178B) [4], the same document suggests formal methods as a suitable alternative for certification [9]. Far from this experiment's intended purpose, this work can be extended to support the certification of FreeRTOS.

Therefore, developing formal verification to a real-time operating system not only increases our confidence in the reliability of the system but it also adds value to the implementation by supporting its certification. The evident value of this kind of work boosts its development.

4.1.5 Pre- vs. post-development

Effectiveness, feasibility, and cost considerations might suggest limiting the application of formal methods only to specific stages of the software development cycle. In the literature, two alternative approaches for applying formal methods have been advocated, each of them with different implications, advantages, and drawbacks:

1. Apply formal methods in the later stages of the development life cycle.

2. Apply formal methods early in the development life cycle.

Proponents of use at the later stages of development argue that it is the final implementation-level design that must be verified. If the final product, be it a piece of software code or a gate-level design or a combination of both, has not been verified, then the verification process is useless.

One of the drawbacks of post-development verification is the lack of a formal description of the system requirements. In case of this particular experiment, documentation in natural language is the only accessible description of FreeRTOS functionality. The translation of the specification from natural language to Separation Logic had to be developed before proceeding with the verification. Had pre-development formal methods been applied, the formal specification would have been accessible and the verification process would have been shorter.

One of the benefits of post-development verification is the possibility to implement formal methods to existing software thanks to the combination of a certified C compiler with an implementation of a formal language to describe the axiomatic semantics of one of its intermediate representations. Furthermore, the architecture design of Compcert makes it possible to replace the current front-end with different implementations. The possibility of developing a compiler front-end that translates a variety of programming languages to Cminor could open the door to applying formal methods for many pieces of software using the same language.

On the other hand, there are several reasons for applying formal methods at early stages. First, verification that the final implementation conforms to its specification is pointless if the specification itself turns out to be flawed and thinking about the specification before implementing it is less likely to taint the specification. Second, it is well known that bugs introduced during the early stages of the design are the most insidious, and fixing them is very expensive when they are discovered late in the development. Finally, proponents of the use of formal methods at the early stages argue that concrete design decisions could better facilitate later development of formal verification.

As an example of this last argument, a research group at Yale University is applying pre-development techniques to develop a certified OS Kernel [49]. Their approach in dealing with the complexity of an OS is the following:

> "to take a clean slate approach to reexamine these different programming concerns and abstraction layers, spell out their formal specifications and invariants, and then design and develop new kernel structures that minimize unwanted interferences and maximize modularity and extensibility". [49]

Another project that makes pre-development verification design decisions to simplify the complexity of the system is SAFE [19]. Thanks to the rapid increase of hardware resources, SAFE can now afford to reconsider some of the traditional sources of complexity in operating systems (i.e. virtual memory, interrupt handling schemes, and manual storage allocation and deallocation) and use simpler design for which proofs are more tractable [19].

4.2 Software Engineering Standpoint

The nature of our experiment has not only verification implications but also software engineering implications. The latter are present in a two fold manner: first, the development of this project roughly followed the development cycle of a software system in that it consisted of design, implementation and documentation phases. Secondly, the implementation of formal methods in the form of creating a formal specification and carrying out the proof of correctness of its implementation impacts the development cycle of the target system. The evaluation of this implications is discussed in the following section.

In the last section, a clear distinction was made between pre- and post-development formal verification to analyze the advantages and disadvantages of one over the other. The purpose of this section is to present a comparison between formal verification versus not implementing formal verification. For this reason, this section makes no distinction between the advantages of both pre- and post-development techniques even though the characteristics of each methodology are clearly distinct.

One of the implicit arguments of this work is that formal verification should be part of the development cycle of safety-critical systems. Taking a broad definition of the software development cycle, we should now discuss how formal verification applies in each phase — design, implementation, documentation, testing, and maintenance.

4.2.1 Design

The experiment presented in this thesis consisted of applying post-development techniques to an existing real-time operating system, which implied working with the C source code developed by FreeRTOS team. It is therefore understood that this experiment did not have an influence on the initial design of the system. This characteristic has the advantage of making it possible to implement formal verification techniques to those systems with a high cost of migration from legacy code to new implementation — a common characteristic of safety-critical systems.

On the other hand, developing a software product with formal verification in mind has many advantages. Consider the cases of using assisted theorem-provers to develop a proof of correctness. The insight of the developer may be of great benefit when completing the proof. In this scenario, the developer would have in mind not only the source code to be written, but also the proof obligations that the implementation entails. To this extent, the design decisions would accommodate for the code to be not only efficient, and easy to read and maintain, but also easier to prove correct.

Another advantage of formal methods, in this case specific to pre-development verification, is that important behaviours of the system can be proven before any code is written. In the case of operating systems, implementations can be lengthy and their design requires an implementation so that properties can be determined empirically [17]. The formal approach will never obviate empirical methods; instead, it allows the designer to state properties of the system *a priori* and to verify them in unambiguous and explicit terms.

A common argument against formal methods is that its use slows down the production of software systems due to its complexity and difficulty. Creating a formal specification of the system certainly increases the time spent during the design phase. However, the additional burden of using formal methods is typically compensated by the higher degree of assurance in the safety and correctness of the system being developed [7]. Furthermore, in some applications, the cost incurred by using formal methods from the early stages of system design are justified by eliminating the significantly higher cost for correcting undetected bugs at later stages.

Formal methods have been used in connection with operating systems for many years; but much of the work has been in the design phase only [17]. An example of this approach can be seen in the modeling of the main functionalities of FreeRTOS by Déharbe et al. [18]. Even though Déharbe does not implement the specified model, it provides a starting point to verify existing implementations of FreeRTOS. Sadly, the majority of the studies in the literature omit the implementation and their proofs.

4.2.2 Implementation

The goal of all software projects is the production of working code. The effects and consequences of the decisions made during the design phase have a direct impact on the implementation of a software system; the benefits of using formal verification techniques are not any different.

Having a specification of the system in a formal language will strongly impact the implementation of the system. A formal specification is the use of notation derived from logic to define the behaviours that the system is to achieve and the assumptions about the world in which the system will operate. In other words, the requirements for system behaviour are explicitly defined. This specification facilitates the understanding of the basic abstraction functionality of the system as well as unambiguously clarifying the implementation decisions. [27, 31]

From personal experience, implementing a system to be proven using the framework presented in this work would require care that termination is easy to prove. One way is to avoid infinite recursion and infinite loops. For example, the implementation of the `vTaskSwitchContext` procedure iterates through the list of tasks by using a counted loop and its termination condition does not consider the index of the list at each iteration but it assumes that the list is not empty. This assumption resulted in more proof obligations, making the proof harder to develop. Moreover, since the implementation of Cminor transforms all loops into guarded loops and Separation Logic forces you to reason in this manner, implementing all the iterative sections as guarded loops as to facilitate reasoning about its behaviour is the preferred strategy.

One can argue that a disadvantage of using formal verification is the increased time it takes to produce working code, resulting in an increase of the cost of development. Nevertheless, it is hard to find evidence to support this argument; there are no empirical studies on the development cost for the same piece of software using both a well-established formal method and a comparable informal method. However, experience on the cost of projects that use formal methods is beginning to accumulate. None of this evidence supports the idea that development costs are higher; if anything, it suggests that the use of formal methods reduces the cost of production. [27]

The argument that formal methods are difficult becomes stronger in the implementation phase since someone must develop the proof of correctness - a task that some people think must be left to highly trained mathematicians. The fact is that the mathematics and the logic used in describing the specifications are straightforward; even though Coq provides the full expressive power of CIC, the specification required for this thesis' experiment required little more than first-order logic taught to computer science juniors. On the other hand, the logic and the mathematics used in developing the proof of correctness are more complicated. One might argue is that developers involved in a formal-methods project can be trained to properly use the tools. Indeed, not everyone in the team has to know how to develop a proof, but there could be a specialized team in charge of carrying out such a task. The goal is that, ultimately, every person with a correct understanding of the system behaviour and its specification should be able to carry out a proof of correctness. To this extent, there has been a substantial improvement in the tools that we have these days and more work is being done to facilitate this task. The benefits of formal methods do not come for free; software developers must make the effort in learning formal methods [7].

4.2.3 Documentation

Formal verification requires the description of the system in terms of a formal language. This description can be used as a communication tool among designers, programmers, and users. As with many other kinds of documentation, a formal specification must describe what things must be done, while hiding the details of how things must be done. This provides an appropriate level of abstraction for software designers to understand the system behaviour without considering the small details related to the chosen implementation. Programmers benefit from having a formal specification because it provides a sort of contract that the implementation must meet. The unambiguous nature of a formal language also reduces the problem of developers interpreting the requirements in a different way other than the one was intended [7]. Finally, the users benefit from the specification by helping them understand the product. As any other learning exercise, it is not until a person tries to explain something that he will know he understands it. Making the system's specification comprehensible to the user can assist the designers to understand the specification because the designer must find a way to paraphrase the specification in natural language. The formal specification does not have to be the end product to explain the system to the user, but the many descriptions, tools, and diagrams that arise from the formal specification can help.

In the specific case of theorem-based formal verification, the documentation of the software product could be extended by adding a sketch of the proof. In other words, we can have documentation of what each theorem means and how its proof was developed. From personal experience, such practice is not very common in existing machine-checked proofs. While it is true that most of the theorems are self-descriptive, the actual proofs of Compcert and Separation Logic are, in many cases, not well-documented. Moreover, the unofficial convention of short and poorly descriptive variable names makes the proof harder to understand. As a relatively new methodology for formal verification using proof assistants, there is room for improvement in the software engineering of proofs.

4.2.4 Testing

In chapter 1, we claimed that testing is insufficient to ensure the reliability of critical systems. A more-rigorous approach using formal verification was proposed as an alternative. At first sight, it might seem that formal verification should entirely replace testing in the development of a software product by augmenting the reliability in the design and implementation phases. Nevertheless, we must be careful in understanding what formal methods guarantee.

It is important to understand the intrinsic limitations of formal methods which arise mainly from the fact that some things can never be proven. A proof is a demonstration that one formal statement follows from another; but, the physical world is not a formal system. A proof, therefore, does not show that things will happen as you expect in the real world [27]. As an example, one of the proofs in this experiment guarantees that the `vTaskSwitchContext` procedure will select the next executing task in a limited amount of time; but, there might be some event in the real world that invalidates this proof. For example, nothing can guarantee that a person will not unplug the cable of the computer on which the system is running.

This does not imply that we should abandon our efforts. All engineering disciplines model the real world and use those models to design artifacts. Models based on mathematics are ideal because the properties of the model can be established by reasoning, and such models can be manipulated during the design. In general, the correspondence between the mathematical models used in the software industry and the real world are well-understood and are used in many other disciplines. This increases our trust in them.

It is also important to acknowledge the kind of properties that a formal system allows us to reason about, and those that are not possible to specify in a formal language. Separation Logic, for example, is an appropriate formal language to describe functional properties; but, it appears impossible to define nonfunctional properties like maintainability and usability.

Applying formal methods, the testing phase can certainly be reduced by eliminating function-related tests like black-box testing, white-box testing, unit testing, and functional testing [32]. Regression testing could also be replaced as long as the system specification did not change when developing the new version. Performance testing is in practice, replaced given that termination properties are provided and the cost analysis has been performed.

Nonfunctional tests, on the other hand, can increase our confidence in the system by testing those properties that are problematic to formalize. Integration testing can be performed to verify the interoperability of the system with those components that have not been formally verified. Usability testing can take its place in verifying that the system is user-friendly. Recovery testing can tell us how well the system recovers from crashes.

In summary, the application of formal methods does not eliminate the need for testing, but it can reduce its duration and increase confidence in the software product. Experience suggests that inspections of formal specifications reveal more errors than those of informal specifications [27] and the end result has been proven to be more reliable compared to those systems that do not apply formal methods [36, 55].

4.2.5 Maintenance

Code maintenance consists of making changes to the software to cope with redesign needs. These needs arise mainly for two reasons [20]:

1. The system does not meet the user needs, in which case corrective maintenance is required.
2. The system undergoes a series of enhancements and extensions as part of adaptive, perfective and preventive maintenance

Formal models reduce code maintenance significantly by eliminating corrective maintenance; once the system's specification has been proven to hold of the system implementation there is no need to make changes to the code.

On the other hand, the need for enhancements and extensions is independent of whether formal methods were used or not. Making use of formal methods helps us in planning code maintenance. As an example, we examine our extension to Compcert to include a tool that calculates the cost of execution of a given program based on the generated assembly code instructions' cycle count.

Adding an extension to Compcert to include the cost analysis tool appeared straightforward. The cost analysis tool would be the last of the compilation phases and would not interfere with any of the other phases in Compcert. The task became more complex when the cost analysis lacked information to provide an accurate estimate when dealing with special cases of function definitions.

Built-in functions are treated differently than non built-in functions, causing the code generated for each to be different. An accurate solution requires each function definition to be either built-in or not. To differentiate between the two variations, Compcert uses the name of the function, which was extended during the parsing phase to include the keyword `__builtin__` for the appropriate cases. This information was lost when moving from the parser to other phases of the compiler. Hence, it was unavailable during the generation of assembly code. Moreover, the parser for Compcert was programmed in Ocaml, making the passing of the information from Ocaml code to Coq code even more difficult.

The initial decision was to make a change to the internal representation of all the compiler's front- and back-end phases to send the name of the function as opposed to sending only the internal identifier for each function definition. The complexity of the task rapidly increased because every use of the internal identifier would have to be replaced affecting a large portion of the code base. Furthermore, many proofs would require (admittedly boilerplate) modifications regarding name lookup. This would not scale.

A second approach consisted of sending a list of function names and identifiers as another variable to each one of the phases by adding an extra argument to each one of the function calls. This argument would simply be ignored in all phases except in the cost analysis. This decision significantly decreased the complexity of the problem[1]; as a result, only five theorems were modified, seven new theorems were required, and one tactic was extended.

Having a formal specification and the proof of correctness for Compcert's implementation required careful decisions about the new cost-analysis module to be added to the compiler. Making changes to a verified system requires not only close attention to not break the code; but, also to maintain the properties of the system behaviour, and their proof of correctness. The biggest advantage of this formalism is the confidence in knowing that even after the modifications, the system will still behave as it was originally designed, since the proof obligations that accompany Compcert were successfully executed after the changes to the compiler were made.

[1] There was not a final estimate on the effort that changing the AST would require, but judging on the high dependency of such data structure in the compiler architecture, it is reasonable to conclude that the effort would have been much larger.

Chapter 5

Summary

To summarize our research, we reflect on the special features of FreeRTOS that enabled our analysis and consider the broader topic of full-featured operating systems, recognize future limitations in the work, and describe future activities.

FreeRTOS is not a full-featured user-level operating system. It supports a restricted class of systems and so its core functions are memory and process isolation. FreeRTOS is designed so that real time applications can be structured as a set of independent tasks. Each task executes within its own context with no coincidental dependency on other tasks within the system or the scheduler itself [39]. According to Tanenbaum [52], two important properties of the memory module have to be provided to allow multiple applications to be in memory at the same time without interfering with each other: protection and relocation.

FreeRTOS provides the solution to protection by the use of an address space — a set of addresses that a process can use to address memory. FreeRTOS' memory manager divides memory into allocation units with the support of an allocation bitmap and a bump pointer, providing a simple way to keep track of memory. Memory protection is enforced by the memory manager by never allocating the same memory space for two or more processes. It is incumbent on the processes to only use memory allocated to them. The experiment in section 3.1.1 provides evidence that the memory protection property can be proved by formally specifying exclusive allocation using Separation Logic and constructing the proof object using Coq proof assistant.

Another way to keep track of memory is to maintain a linked-list of allocated and free memory segments, where a segment either is assigned to a process or the segment is free [52]. FreeRTOS provides an alternative memory module that makes use of this scheme to manage free memory. Even though this work does not provide the proof of that module's correct implementation, the tools used in the experiment are perfectly suitable to carry out such proof. As an example, the implementation of Separation Logic by McCreight [41] proves the expressiveness of separation logic and the ease of use of his tactics by showing the proof of correctness of an in-place list reversal program. In other words, Separation Logic is suitable to express data structures such as lists and coupled with our proofs, a chunked memory allocator can be verified.

Moving to the second property, there exist two general approaches to dealing with memory overload to solve the relocation problem : memory swapping and virtual memory [52]. Memory swapping consists of bringing in each process in its entirety, running it for a while, then putting it back to the external storage. Virtual memory allows programs to run even when they are only partially in main memory. Either of these approaches is appropriate when the physical memory of the system is not large enough to hold all the processes.

The FreeRTOS real-time kernel has been designed specifically for small, embedded systems [39] for which it is expected that memory is large enough to hold all the processes simultaneously, not needing a solution for relocation at all. This expectation is borne out for a wide variety of control systems. For example, the avionics system in the F-22 Raptor — the current U.S. Air Force frontline jet fighter — consists of about 1.7 million lines of software code. The F-35 Joint Strike Fighter requires about 5.7 million lines of code. And Boeing's 787 Dreamliner requires about 6.5 millions lines of code [11]. To get an idea of the size of the executable binary code consider the Linux kernel version 2.4.0 with a source code of nearly 3.38 million lines of code and a compiled binary size of about 1.1MB [47]. None of these avionics systems would be considered a small system, yet an ARM7 processor (one of the many architectures supported by FreeRTOS) could easily hold a project of this magnitude.

By presenting evidence of the formal verification of FreeRTOS protection mechanism and demonstrating that thanks to the rapid increase of hardware resources a relocation mechanism is not essential for the target systems for which FreeRTOS was designed, it is fair to conclude that the significant properties of FreeRTOS memory manager can be described formally using Separation Logic and its verification can be developed using the Coq proof assistant.

Another key component of an OS is process scheduling. The FreeRTOS scheduler is the part of the kernel responsible for deciding which task should be executing at any particular time. Since real-time systems are designed to provide a timely response to real world events, the foremost need in a real-time scheduling mechanism is the ability to meet all the deadlines [52]. FreeRTOS enforces the achievement of all deadlines by establishing a static scheduling policy based on priorities; the scheduling mechanism is then in charge of ensuring that the highest priority task that is able to execute is the task that is given processing time [39]. Therefore, to characterize FreeRTOS scheduling mechanism as correct, we must guarantee that the system will meet all deadlines and that the scheduling policy is enforced.

The ability of a real-time system to meet all its deadlines depends upon the number of events it has to respond to and the amount of time it takes to process each event. For example, if there are m periodic events and event i occurs with period P_i and requires C_i seconds to finish its processing, then the scheduler will meet all its deadlines if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

Even when the number of tasks is not known beforehand, knowing the time it takes for each task to complete can help to determine the maximum number of tasks the system can safely schedule. As an example, consider a system with three types of events: two periodic events (a *and* b) with periods of 100 and 200 ms respectively and one event (c) with unknown period. If these events require 50, 30, and 100 ms response time respectively then we know that the system can only serve an event c every 286 ms not allowing for scheduling overhead. In other words, knowing the time of execution of each task is critical in the design of real-time systems.

Calculating the time of execution of each task requires the guarantee that the task will terminate. Despite the obvious nature of this condition, it is possible to create tasks that will never finish their execution. For this reason, care must be taken to make sure the tasks involved in a real-time operating system always terminate.

In section 3.1.2 different paths that lead to nontermination were discussed and a proof was provided to show that the `vTaskSwitchContext` operation has this property. This exercise is certainly not complete in the sense that it does not cover every part of the operating system, but the same methods can be applied to other operations inside the scheduler module and different task-oriented modules comprising FreeRTOS.

Moreover, an extension to Compcert was provided that calculates the cost of execution based on the cycle count of the assembly code generated by the compiler. Using this tool, it is possible to estimate the execution time of each part of a full FreeRTOS implementation.

In other words, the experiment that shows that the scheduling mechanism will select the correct next task to execute in a fixed amount of time along with the tool to calculate the cost of execution provides enough evidence to verify that FreeRTOS scheduler will meet its deadlines, and an entire system will operate correctly.

Termination of the `vTaskSwitchContext` procedure was shown because this proof can be extended to show that FreeRTOS scheduler also enforces its scheduling policy: the task with the highest priority that is ready to be executed is the one that will be granted processing time. The proof in appendix C shows that the iterative operations inside the `vTaskSwitchContext` procedure will terminate in a given amount of time. Such iterative operations are part of the procedure in charge of selecting the highest priority task. Adding the rest of the commands to complete the process and providing its proof can be done following the same methodology as shown in proving the safety properties of FreeRTOS memory module.

By presenting evidence that the scheduler will meet the deadlines in the form of proving its termination and its time of execution, one of the important properties of a real-time scheduling mechanism has been covered as a proof of concept. Moreover, this experiment provides a starting point to show that FreeRTOS scheduler enforces its scheduling policy. In other words, the properties that characterize a reliable scheduling mechanism can be expressed in Separation Logic terms and its formal verification can be developed using the Coq proof assistant.

In conclusion, after expressing the significant properties of FreeRTOS in a formal language, providing evidence of its formal verification using Coq proof assistant, and describing how this approach can be extended to verify the correctness of application-specific FreeRTOS parts, the current state-of-the-art in theorem-based formal verification makes it possible to provide a machine-checked proof of the formal specification of a real-time operating system.

5.1 Limitations

Throughout the development of this experiment some limitations were encountered that range from the strength of the proofs provided to the expressiveness of the tools used in this experiment. In this section some of these limitations are explored.

5.1.1 Guaranteeing memory exclusive access

The proof of FreeRTOS memory management system was discussed in section 3.1.1. One of the proven properties carried out in this experiment was that the memory manager will provide exclusive allocation of memory among different processes. A proof that guarantees exclusive access requires more than the correct behaviour of the memory manager; it requires either

1. a memory protection mechanism
2. that all the processes are trusted to honour exclusive access.

The solution of a memory protection mechanism is provided by FreeRTOS by a Memory Protection Unit (MPU) module, but its scope is limited. Using an MPU can protect applications from a number of potential errors, ranging from undetected programming errors to errors introduced by system or hardware failures. A FreeRTOS MPU can also be used to protect the kernel itself from invalid executions by tasks and protect its data from corruption. It can also protect system peripherals from unintended actions by tasks and guarantee the detection of stack overflows.

Showing that an implementation of the FreeRTOS MPU is correct with respect to its specification would provide stronger evidence about the exclusive access property required of a reliable system. However, a FreeRTOS MPU has a limited scope, since it is specified for specific ARM hardware, the Cortex-M3 micro-controllers. In case FreeRTOS is targeted to a different architecture, a different solution is required.

The second way exclusive access can be guaranteed is by showing that all the tasks running on the system are trusted. Not having different modes of execution (i.e. *user-mode* and *kernel-mode*) weakens the safety of FreeRTOS since nothing stops user-mode tasks from bypassing the operating system and accessing any memory location, including those in other tasks. Therefore, any task running on FreeRTOS needs to supply a proof of its compliance with the policies demanded by the operating system. In other words, the tasks running in FreeRTOS must meet the following properties [17]:

1. They are trusted to honour memory exclusive access. (safety properties)
2. Their behaviour is entirely predictable (deterministic/liveness properties)
3. Their behaviour won't cause the system to miss the deadlines (they run for relatively shorts periods of time when executed)

Consequently, to increase the reliability of FreeRTOS, every task that is to run in the system must supply proofs meeting these obligations as a minimum requirement to categorize them as reliable. A more complete solution would also include the proof that the functional requirements of each task are met by the implementation.

5.1.2 Inline assembly instructions

As the interface layer between hardware and user processes, real-time operating systems are highly coupled hardware devices. For this reason, the implementation of an operating system usually requires the use of *inline assembly code* — low-level constructs to control registers and access hardware directly.

With more than 32 supported architectures, portability is one of the prime concerns in the design of FreeRTOS. Because of this, the implementation of FreeRTOS provides much architecture-independent code. Nevertheless, parts of the code are inherently dependent upon the architecture and the use of inline assembly code is not uncommon. The code in charge of enabling and disabling hardware interrupts is an example of code that contains inlined assembly instructions.

As discussed in section 2.2.4, Separation Logic serves as a logic to formally define the axiomatic semantics of Cminor. Thanks to Separation Logic, we can reason about Cminor programs by describing the state of the program before and after each Cminor instruction. That is, the formal description of the properties of FreeRTOS memory manager and scheduler were possible in large part thanks to this logic. Unfortunately, Separation Logic for Cminor does not allow us to reason about lower level constructs (i.e. inlined assembly code). Hence, Separation Logic is not suitable for an exhaustive analysis of FreeRTOS code without an extension to include this type of constructs.

A plausible solution to this limitation would be to implement Separation Logic for assembly language and carry out all the proofs in this lower level language. This would require a much higher level of complexity due to the higher number of instructions per statement and the lack of familiarity of the programmer with assembly code.

Another solution could be the use of other formal methods such as model checking. Since the complexity of these simpler assembly operations (i.e. enabling hardware interrupts) is lower than the complexity of a full operating system, the approach to formal verification using model checking for assembly fragments could be effective. The final full FreeRTOS validation would then consist of a combination of properties carried out using theorem-proving techniques and some other properties proven by model checking.

5.1.3 Coq and the use of *large* numbers

In Coq's programming language, almost nothing is built-in - not even booleans or numbers [46]. Instead, Coq provides powerful tools for defining new types of data and functions that process and transform them.

The way Coq deals with numbers is by using the Morgenson-Scott encoding of Church numerals built from the numeral 0 and the successor function `S`. Even when numbers can be denoted using decimal notation, Coq's internal representation uses a linear structure to actually build the number. (i.e. when the notation `3%nat` is used, Coq internal representation is `S (S (S 0))`. A drawback of this approach is the inefficiency of computing with numbers. Even more so, behind the scene, Coq will have to build a proof object that increases in size as the numbers used in the computation grow larger.

The use of large numbers turned out problematic when proving the safety property of the memory manager to guarantee that it will never try to allocate more memory than available. This proof required to show that the memory manager will always stay within the range of `configTOTAL_HEAP_SIZE`. The default value for the heap size in an ARM7 architecture provided by FreeRTOS is 22,000 bytes. Trying to carry out the proof with this value resulted in waiting periods in the order of hours waiting for Coq to compute the comparison between two numbers.

This behaviour soon became inadequate and the decision was made to decrease the size of the heap to a lower value (i.e. 100) to speed up the response time of Coq. The argument behind this decision is that such change does not alter the correctness of the system and that the proof can be extended to use the actual value for the size of the heap; the proof of correctness using the actual value for the size of the heap can still be produced given the right amount of time to wait for the proof object.

5.2 Future Work

Reducing, and potentially eliminating, the limitations found in this experiment is future work. In addition, there are other areas of opportunity to expand the work shown in this thesis including:

1. Providing proofs of more safety and liveness properties for FreeRTOS

2. Verify the correct implementation of the underlying structures of FreeRTOS

3. Extend FreeRTOS microkernel to include other kernel components such as file system, network protocols and device drivers along with their respective proof of correctness

4. Increase the level of automation of the tools used

These areas of opportunity are discussed in this section.

The next logical step is to extend this work with the proof of other properties that will increase the trustworthiness of FreeRTOS. This experiment consisted of proving the significant properties of the memory module and scheduler of FreeRTOS; while these properties are considered to be essential for the correct behaviour of a real-time operating system they fall short from being complete.

Craig [17] proposes a high-level specification of a real-time operating system that includes not only the properties discussed in this thesis, but also requires a specific behaviour for individual processes and their state. For example, Craig suggests that a trusted process must be in exactly one state in a given time. In other words, each process can only be in either a *ready* state, *running* state, *waiting* state or *terminated* state. Moreover, Craig suggests that it is required to guarantee that when processes are not executing, they do nothing. This implies that processes cannot make requests to devices, nor can they engage in inter-process communication or any other operations that might change multiple process' state.

While these properties are required by FreeRTOS, there is no formal proof that guarantees this behaviour for individual processes. On the other hand, it is possible to provide formal evidence of this behaviour by following the methodology discussed in this experiment.

Another possible extension to this work is the proof of the correct behaviour of the underlying structures encountered in operating system kernels such as *queues* and *lists*. The proofs provided in this thesis rely on the behaviour of the underlying data structures. For example, the proof of termination for the `vTaskSwitchContext` procedure relies on the queue of ready tasks to behave as a FIFO data structure.

The decision to not validate the correct implementation of the underlying data structures is based on the idea that such data structures are simple enough that their behaviour is well understood and their implementation is uncomplicated. While this argument allowed me to focus on the high level specification of a real-time operating system, a more complete commercial implementation could include proofs of the correct behaviour of the underlying structures.

The FreeRTOS microkernel could also be extended to include other kernel components such as device drivers, file systems, and network protocols to provide a more complete system. The portability of FreeRTOS comes with a price that reduces its readiness to be deployed as a system in itself; much of the work for a complete solution is left to the programmers to create the tasks that will execute most of the computation and interaction with the controlled system.

It is from the interaction with the controlled system that the implementation of input/output device drivers would be the most beneficial in extending FreeRTOS. As with any other user-created task running in FreeRTOS, to characterize the implementation of these device drivers as correct, these would have to be accompanied by proof objects that guarantee their behaviour as reliable (as outlined in section 5.1.1) . Even more so, the correctness properties would have to be devised and the evidence in the form of proof object would have to be present, possibly in a similar manner as how this experiment approached the verification of a real-time operating system.

Finally, the level of automation of the tools used in this experiment can be increased. One way this can be achieved is by the annotation of the C code with its formal specification in Separation Logic by the use of the pre- and postconditions in the form of comments, and then extracting the specification as part of the transformation from C to Cminor. This would significantly reduce the time spent in setting up the definitions required to carry out the proofs and would allow the programmer to focus his efforts in discharging the proofs.

The level of automation for the discharge of the proof can also be increased by the use of recent tools such as Bedrock [13]. Bedrock is a framework for implementation and verification of low-level programs in Coq that provides a mostly-automated discharge of Separation Logic verification conditions. As an example of the increased level of automation of Bedrock when compared to the Separation Logic by McCreight's implementation used in this experiment, the verification of an in-place list reversal function has been carried out. McCreight's proof includes about 80 atomic tactic calls while the Bedrock proof includes about 10 atomic tactic calls.

References

[1] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step Cminor. In *Theorem Proving in Higher Order Logics '07*, volume 4732 of *Lecture Notes in Computer Science*, pages 5–21, Kaiserslautern, Germany, 2007. Springer.

[2] Tamarah Arons, Elad Elster, Limor Fix, Sela Mador-Haim, Michael Mishaeli, Jonathan Shalev, Eli Singerman, Andreas Tiemeyer, Moshe Vardi, and Lenore Zuck. Formal verification of backward compatibility of microcode. In *Computer Aided Verification '05*, volume 3576 of *Lecture Notes in Computer Science*, pages 93–134, Edinburgh, Scotland, UK, 2005. Springer.

[3] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A decade of software model checking with SLAM. *Communications of the ACM*, 54(7):68–76, July 2011.

[4] Richard Barry. Compiler verification for safety-critical applications. *Embedded Systems Design Europe*, 6:32–35, June - July 2007.

[5] William R. Bevier. *A Verified Operating System Kernel*. Technical report, University of Texas, 1987.

[6] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *Formal Methods '06*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475, Hamilton, Ontario, Canada, 2006. Springer.

[7] Marco Bozzano and Adolfo Villafiorita. *Design and Safety Assessment of Critical Systems*, chapter Formal Methods for Safety Assessment, pages 139–212. Auerbach Publications, 2010.

[8] Marco Bozzano and Adolfo Villafiorita. *Design and Safety Assessment of Critical Systems*, chapter Formal Methods for Certification, pages 213–239. Auerbach Publications, 2010.

[9] Duncan Brown, Herve Delseny, Kelly Hayhurst, and Virginie Wiels. Guidance for using formal methods in a certification context. In *Embedded Real Time Software and Systems '10*, Toulouse, France, May 2010.

[10] Sam Buss. Weak formal systems and connections to computational complexity. Lecture Notes, University of California Berkley, 1988.

[11] Robert N. Charette. This car runs on code. *IEEE Spectrum*, 46(2), February 2009.

[12] Adam Chlipala. *Certified Programming with Dependent Types*. http://adam.chlipala.net/cpdt/, 2009.

[13] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI '11*, pages 234–245, San Jose, California, USA, 2011. ACM.

[14] Edmund Clarke. The birth of model checking. In *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 1–26, Seattle, Washington, USA, 2008. Springer.

[15] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, February 1988.

[16] P. Cousot. *Methods and Logics for Proving Programs*, volume B of *Handbook of Theoretical Computer Science*, chapter 15, pages 843–993. Elsevier Science Publishers, 1990.

[17] Iain D. Craig. *Formal Models of Operating System Kernels*. Springer, 2007.

[18] David Déharbe, Stephenson Galvão, and Anamaria Martins Moreira. *Formal Methods: Foundations and Applications*, chapter Formalizing FreeRTOS: First Steps, pages 101–117. Springer, 2009.

[19] André DeHon, Ben Karel, Thomas F. Knight, Jr., Gregory Malecha, Benoît Montagu, Robin Morisset, Greg Morrisett, Benjamin C. Pierce, Randy Pollack, Sumit Ray, Olin Shivers, Jonathan M. Smith, and Gregory Sullivan. Preliminary design of the SAFE platform. In *PLOS '11*, pages 4:1–4:5, Cascais, Portugal, 2011. ACM.

[20] B. S. Dhillon, editor. *Software Maintenance*, chapter Software Maintenance. CRC Press, 2002.

[21] William R. Dunn. Designing safety-critical computer systems. *IEEE Computer*, 36(11):40–46, November 2003.

[22] Raphael Finkel, Thomas Anderson, Brian Bershad, Edward Lazowska, Henry Levy, Robert Cupper, John Stankovic, Craig Wills, Peter Denning, Marshall Kirk McKusick, Sape Mullender, Steve Chapin, Jon Weissman, and T Doeppner. *Operating Systems*, chapter What Is an Operating System? Chapman and Hall / CRC, 2004.

[23] Limor Fix. Fifteen years of formal property verification in Intel. In *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 139–144, Seattle, Washington, USA, 2008. Springer.

[24] Carlo Ghezzi, Mehdi Jazayeri, Dino Mandrioli, Ian Sommerville, Steven Demurjian, Patricia Pia, Gregory Kapfhammer, Jonathan Bowen, Michael Hinchey, John Gannon, Roger Pressman, Stephen Seidman, Osama Eljabiri, and Fadi Deek. *Software Engineering*, chapter Formal Methods. Chapman and Hall / CRC, 2004.

[25] Carlo Ghezzi, Mehdi Jazayeri, Dino Mandrioli, Ian Sommerville, Steven Demurjian, Patricia Pia, Gregory Kapfhammer, Jonathan Bowen, Michael Hinchey, John Gannon, Roger Pressman, Stephen Seidman, Osama Eljabiri, and Fadi Deek. *Software Engineering*, chapter Software Testing. Chapman and Hall / CRC, 2004.

[26] J. Gray and D.P. Siewiorek. High-availability computer systems. *IEEE Computer*, 24(9):39–48, September 1991.

[27] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990.

[28] Robert Harper. *Practical Foundations for Programming Languages*. http://www.cs.cmu.edu/~rwh/plbook/book.pdf, April 2011.

[29] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, October 1969.

[30] Michael Huth and Mark Ryan. *Logic in Computer Science: modelling and reasoning about systems 2nd ed.* Cambridge University Press, 2004.

[31] Sally Johnson and Ricky Butler. *Electrical Engineering Handbook*, chapter Formal Methods. CRC Press, 2000.

[32] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, and Armaghan Naik. Replacing testing with formal verification in Intel core i7 processor execution engine validation. In *Computer Aided Verification '09*, volume 5643 of *Lecture Notes in Computer Science*, pages 414–429, Grenoble, France, 2009. Springer.

[33] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *SOSP '09*, pages 207–220, Big Sky, Montana, USA, 2009. ACM.

[34] J.C. Knight. Safety critical systems: challenges and directions. In *ICSE '02*, pages 547–550, Orlando, Florida, USA, May 2002.

[35] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, March 1977.

[36] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[37] Nancy G. Leveson. An investigation of the Therac-25 accidents. *IEEE Computer*, 26:18–41, 1993.

[38] ARM Limited. *ARM7TDMI Technical Reference Manual 4th ed.* ARM Limited, November 2004.

[39] Real Time Engineers Ltd. *The FreeRTOS Project Version 6.1.0.* http://freertos.org, October 2010.

[40] Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. Formal verification of the heap manager of an operating system using separation logic. In *International Conference on Formal Engineering Methods '06*, volume 4260 of *Lecture Notes in Computer Science*, pages 400–419, Macau, China, 2006. Springer.

[41] Andrew McCreight. Practical tactics for separation logic. In *Theorem Proving in Higher Order Logics '09*, volume 5674 of *Lecture Notes in Computer Science*, pages 343–358, Munich, Germany, 2009. Springer.

[42] Ann Marie Neufelder. *Ensuring Software Reliability*, chapter Factors that Affect Software Reliability, pages 45–64. CRC Press, 1992.

[43] Ann Marie Neufelder. *Ensuring Software Reliability*, chapter Defining Software Reliability, pages 9–20. CRC Press, 1992.

[44] Peter G. Neumann. *Computer-Related Risks.* Addison-Wesley, 1995.

[45] Peter G. Neumann. Illustrative risks to the public in the use of computer systems and related technology. *SIGSOFT Software Engineering Notes*, 21(1):16–30, January 1996.

[46] Benjamin C. Pierce, Chris Casinghino, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjoberg, and Brent Yorgey. *Software Foundations.* http://www.cis.upenn.edu/~bcpierce/sf/, July 2012.

[47] The Linux Information Project. *Source Code Definition.* http://www.linfo.org/source_code.html, February 2006.

[48] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science '02*, pages 55–74, Copenhagen, Denmark, 2002. IEEE Computer Society.

[49] Z. Shao and B. Ford. *Advanced Development of Certified OS Kernels.* Technical report, Yale University, New Haven, CT, USA, July 2010.

[50] John Stankovic. *Operating Systems 2nd ed.*, chapter Real-Time and Embedded Systems. Chapman and Hall / CRC, 2004.

[51] John A. Stankovic and R. Rajkumar. Real-time operating systems. *Real-Time Systems*, 28:237–253, November - December 2004.

[52] Andrew S. Tanenbaum. *Modern Operating Systems 3rd ed.* Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[53] Coq D. Team. *The Coq proof assistant reference manual, version 8.2.* http://coq.inria.fr/V8.2pl3/files/Reference-Manual.pdf, August 2009.

[54] Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):118–131, February 1980.

[55] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *PLDI '11*, pages 283–294, San Jose, California, USA, 2011. ACM.

Appendix A

Library Memory_Safety_1

```
(*  This file contains the proof that FreeRTOS memory allocation procedure
   won't allocate more memory than available as it is described in section
   3.1.2 Safety Properties of FreeRTOS.*)

(*  Import Seperation Logic Libraries *)
Require Import ProgLog.
Require Import NatUtil.
Require Import Addr.

Set Implicit Arguments.

(*  Define global variables *)
Definition xws := Var 0.
Definition vTaskSuspendAll := Var 1.
Definition xTaskResumeAll := Var 2.
Definition xNextFreeByte := Var 3.
Definition xHeap := Var 4.
Definition configTOTAL_HEAP_SIZE := 100.

(*  Define local variables *)
Definition a1 := Var 5.
Definition a2 := Var 6.
Definition pvReturn := Var 7.
Definition aux1 := Var 8.
Definition aux2 := Var 9.

(*  Malloc
   The function allocates required memory and returns a pointer
   to the first free location
   *)
Notation mallocBody :=
  ((pvReturn ← 0);;
    (call aux1 (euop ldop vTaskSuspendAll) nil);;
    (ifte (ebop ltop (ebop plusop (euop ldop xNextFreeByte) (euop ldop xws)) 100%val)
      ((ifte (ebop gtop (ebop plusop (euop ldop xNextFreeByte) (euop ldop xws)) (euop ldop xNextFreeByte))
        (a1 ← 1)
        (a1 ← 0));;
      (a2 ← a1))
      (a2 ← 0));;
    (ifte (a2 != 0)
      ((vassign pvReturn (ebop plusop (ebop multop 1%val (euop ldop xNextFreeByte)) (euop ldop xHeap)));;
        (store xNextFreeByte (ebop plusop (euop ldop xNextFreeByte) (euop ldop xws))))
      (skip));;
    (call aux2 (euop ldop xTaskResumeAll) nil);;
    ret pvReturn)%CM.

(*  The definition of the program, including its global and local variables *)
Notation mallocDef :=
  (fdefi (xws::vTaskSuspendAll::xTaskResumeAll::xNextFreeByte::xHeap::nil) (a1::a2::pvReturn::aux1::aux2::nil)
0 mallocBody).
(*  The type of the arguments passed to the program *)
(*  In detail:
        xws is an integer representing the size of the memory to be allocated
        vTaskSuspendAll is the address where such function can be found
        xTaskResumeAll is the address where such function can be found
        xNextFreeByte is the value of the next free memory location
        xHeap is pointing to the head of the memory "heap" *)
Notation mallocTy := (val::addr::addr::val::val::nil :tslist).
(*  The precondition that has to be met in onder for the program to hold
   the postcondition *)
(*  The precondition states that the current value of xNextFreeByte is less
   than the total size of the heap and that the addresses and pointers are not null *)
Definition mallocPre a (b c: addr) d e u w x y z :=
  lexists s, lexists s’, lexists s”, lexists s”’,
  u |-> a ** w |-> b ** x |-> c ** y |-> d ** z |-> e **
    !(trueVal (bopVal ltop d configTOTAL_HEAP_SIZE)) ** !(a = Vword s) ** !(b = (s’,0)%addr) ** !(c
= (s”,1%word)) ** !(d = Vword s”’).
(*  The desired final state after the program has been executed *)
(*  The postcondition states that after finishing execution, the value of
   xNextFreeByte will be within the range of the total size of the memory *)
Definition mallocPost a (b c: addr) (d e u w x y z _ : val) :=
  lexists p,
  u |-> a ** w |-> b ** x |-> c ** y |-> p ** z |-> e ** !(trueVal (bopVal ltop p configTOTAL_HEAP_SIZE)).
(*  PROVIDING XTASKSUSPENDALL SPECIFICATION *)
Definition suspendTy := (nil : tslist).
Definition suspendPre := emp.
Definition suspendPost ( _ : val) := emp.
Definition suspendSpec := speci suspendTy 0%nat suspendPre suspendPost.
(*  PROVIDING XTASKRESUMEALL SPECIFICATION *)
Definition resumeTy := (nil : tslist).
Definition resumePre := emp.
Definition resumePost ( _ : val) := emp.
Definition resumeSpec := speci resumeTy 0%nat resumePre resumePost.
(*  PROVIDING CODE HEAP TYPE *)
Definition P1 : cdhpty := fun c ⇒
  let (b, w) := c in
    match beq_nat (word_to_nat w) 0 with
      | true ⇒ Some suspendSpec
      | false ⇒ Some resumeSpec
    end.
(*  The theorem stating that the desired final state will hold true of the program
   given that the preconditions are met *)
Lemma mallocOk : fdefOk P1 mallocTy 5%nat mallocPre mallocDef mallocPost.
Proof.
  fdefBegin. unfold mallocPre, mallocPost. intros a b c d e u w x y z m sp Hp vf VFE.
  ssimpl in Hp. simpl in VFE.
  vcSteps.
```

```
subst b.
autorewrite with ProgLog.
∃ nil.
unfold P1.
rewrite beq_true.
unfold suspendSpec.
unfold suspendTy.
callStep.
∃ (u |-> a ** w |-> x1 ** x |-> c ** y |-> d ** z |-> e).
simpl addArgs.
unfold someBind.
constructor.
searchMatch.
intros v' m' Hs.
vfUpdateStep VFE aux1 v'; dropVfeDups VFE.
basicStmVCStep.
bigStmVCStep.
branchStep.
branchStep.
branchStep.
∃ nil.
subst c.
rewrite beq_false.
unfold resumeSpec.
unfold resumeTy.
callStep.
∃ (u |-> a ** w |-> x1 ** x |-> (addrAdd x2 1) ** y |-> (d%val + a%val)%val ** z |-> e).
simpl addArgs.
unfold someBind.
constructor.
searchMatch.
simpl addrAdd.
replace (0 + 1)%word with (1).
reflexivity.
womega.

intros v'0 m'0 Hs'.
vfUpdateStep VFE aux2 v'0; dropVfeDups VFE.

vcSteps.
ssimpl.
searchMatch.
simpl addrAdd.
replace (0 + 1)%word with (1).
reflexivity.
womega.

assumption.

discriminate.

branchStep.
∃ nil.
subst c.
rewrite beq_false.
```

```
unfold resumeSpec.
unfold resumeTy.
callStep.
∃ (u |-> a ** w |-> x1 ** x |-> (addrAdd x2 1) ** y |-> d ** z |-> e).
simpl addArgs.
unfold someBind.
constructor.
searchMatch.
simpl addrAdd.
replace (0 + 1)%word with (1).
reflexivity.
womega.

intros v'0 m'0 Hs'.
vfUpdateStep VFE aux2 v'0; dropVfeDups VFE.

vcSteps.
ssimpl.
searchMatch.
simpl addrAdd.
replace (0 + 1)%word with (1).
reflexivity.
womega.

assumption.

discriminate.

subst d.
subst a.
assert (impl (bopVal gtop (Vword (x3 + x0)%word) (Vword x3) = Vundef) False) as H2.
apply vundefWordBopVal with (b := gtop) (w := (x3 + x0)%word) (w' := x3).
unfold impl in H2.
apply H2.
assumption.

branchStep.
∃ nil.
subst c.
rewrite beq_false.
unfold resumeSpec.
unfold resumeTy.
callStep.
∃ (u |-> a ** w |-> x1 ** x |-> (addrAdd x2 1) ** y |-> d ** z |-> e).
simpl addArgs.
unfold someBind.
constructor.
searchMatch.
simpl addrAdd.
replace (0 + 1)%word with (1).
reflexivity.
womega.

intros v'0 m'0 Hs'.
vfUpdateStep VFE aux2 v'0; dropVfeDups VFE.

vcSteps.
```

```
  ssimpl.
  searchMatch.
  simpl addrAdd.
  replace (0 + 1)%word with (1).
  reflexivity.
  womega.

  assumption.

  discriminate.

  subst d.
  subst a.
  assert (impl (bopVal ltop (Vword (x3 + x0)%word) (Vword 100) = Vundef) False) as H2.
  apply vundefWordBopVal with (b := ltop) (w := (x3 + x0)%word) (w' := 100).
  unfold impl in H2.
  apply H2.
  assumption.
Qed.
```

Appendix B

Library Memory_Safety_2

```
(*  This file contains a proof that FreeRTOS memory allocation procedure
   guarantees exclusive memory allocation, enforcing memory protection
   among processes as it is described in section 3.1.1 Safety Properties
   of FreeRTOS.

   Memory exclusive allocation is proved by showing that once a range of
   memory has been allocated for a given process, the memory allocation
   procedure won't allocate a memory location inside such range to a
   different process. *)
(*  Import Separation Logic libraries *)
Require Import ProgLog.
Require Import NatUtil.
Require Import Addr.
Set Implicit Arguments.
(*  Define global variables *)
Definition xws := Var 0.
Definition vTaskSuspendAll := Var 1.
Definition xTaskResumeAll := Var 2.
Definition xNextFreeByte := Var 3.
Definition xHeap := Var 4.
Definition configTOTAL_HEAP_SIZE := 100.
(*  Define local variables *)
Definition a1 := Var 5.
Definition a2 := Var 6.
Definition pvReturn := Var 7.
Definition aux1 := Var 8.
Definition aux2 := Var 9.
(*  Malloc
   The function allocates required memory and returns a pointer
   to the first free location
   *)
Notation mallocBody :=
  ((pvReturn ← 0);;
    (call aux1 (euop ldop vTaskSuspendAll) nil);;
    (ifte (ebop ltop (ebop plusop (euop ldop xNextFreeByte) (euop ldop xws)) 100%val)
      ((ifte (ebop gtop (ebop plusop (euop ldop xNextFreeByte) (euop ldop xws)) (euop ldop xNextFreeByte))
        (a1 ← 1)
        (a1 ← 0));;
      (a2 ← a1))
      (a2 ← 0));;
    (ifte (a2 != 0)
      ((vassign pvReturn (ebop plusop (ebop multop 1%val (euop ldop xNextFreeByte)) (euop ldop xHeap)));;
        (store xNextFreeByte (ebop plusop (euop ldop xNextFreeByte) (euop ldop xws))))
      (skip));;
    (call aux2 (euop ldop xTaskResumeAll) nil);;
    ret pvReturn)%CM.
(* The definition of the program, including its global and local variables *)
Notation mallocDef :=
  (fdefi (xws::vTaskSuspendAll::xTaskResumeAll::xNextFreeByte::xHeap::nil) (a1::a2::pvReturn::aux1::aux2::nil)
0 mallocBody).
(* The definition of what it means for a memory location to not have been
   previously allocated *)
Definition notPreviouslyAllocated (p d : val) :=
  trueVal (bopVal gtop p d) ∨ trueVal (bopVal eqop d p).
(* The type of the arguments passed to the program *)
(* In detail:
        xws is an integer representing the size of the memory to be allocated
        vTaskSuspendAll is the address where such function can be found
        xTaskResumeAll is the address where such function can be found
        xNextFreeByte is the value of the next free memory location
        xHeap is pointing to the head of the memory "heap" *)
Notation mallocTy := (val::addr::addr::val::val::nil :tslist).
(* The precondition that has to be met in onder for the program to hold
   the postcondition *)
(* The precondition states that the current value of xNextFreeByte is less
   than the total size of the heap and that the addresses and pointers are not null *)
Definition exclusionPre a (b c: addr) d e u w x y z :=
  lexists s, lexists s', lexists s", lexists s"',
  u |-> a ** w |-> b ** x |-> c ** y |-> d ** z |-> e **
    !(trueVal (bopVal ltop d configTOTAL_HEAP_SIZE)) ** !(a = Vword s) ** !(b = (s',0)%addr) ** !(c
= (s",1%word)) ** !(d = Vword s"').
(* The desired final state after the program has been executed *)
(* The postcondition states that after finishing execution, the allocated memory
   has not been previously allocated *)
Definition exclusionPost (a b c d e u w x y z rv : val) :=
  lexists p,
  u |-> a ** w |-> b ** x |-> c ** y |-> p ** z |-> e ** !(notPreviouslyAllocated p d).
(* PROVIDING XTASKSUSPENDALL SPECIFICATION *)
Definition suspendTy := (nil : tslist).
Definition suspendPre := emp.
Definition suspendPost ( _ : val) := emp.
Definition suspendSpec := speci suspendTy 0%nat suspendPre suspendPost.
(* PROVIDING XTASKRESUMEALL SPECIFICATION *)
Definition resumeTy := (nil : tslist).
Definition resumePre := emp.
Definition resumePost ( _ : val) := emp.
Definition resumeSpec := speci resumeTy 0%nat resumePre resumePost.
(* PROVIDING CODE HEAP TYPE *)
Definition P1 : cdhpty := fun c ⇒
  let (b, w) := c in
    match beq_nat (word_to_nat w) 0 with
```

```
        | true ⇒ Some suspendSpec
        | false ⇒ Some resumeSpec
      end.
(*  The theorem stating that the desired final state will hold true of the program
   given that the preconditions are met *)
Lemma exclusionOk : fdefOk P1 mallocTy 5%nat exclusionPre mallocDef exclusionPost.
Proof.
  fdefBegin. unfold exclusionPre, exclusionPost. intros a b c d e u w x y z m sp Hp vf VFE.
  ssimpl in Hp. simpl in VFE.
  vcSteps.
  subst b.
  autorewrite with ProgLog.
  ∃ nil.
  unfold P1.
  rewrite beq_true.
  unfold suspendSpec.
  unfold suspendTy.
  callStep.
  ∃ (u |-> a ** w |-> x1 ** x |-> c ** y |-> d ** z |-> e).
  simpl addArgs.
  unfold someBind.
  constructor.
  searchMatch.
  intros v' m' Hs.
  vfUpdateStep VFE aux1 v'; dropVfeDups VFE.
  basicStmVCStep.
  bigStmVCStep.
  branchStep.
  branchStep.
  branchStep.
  ∃ nil.
  subst c.
  rewrite beq_false.
  unfold resumeSpec.
  unfold resumeTy.
  callStep.
  ∃ (u |-> a ** w |-> x1 ** x |-> (addrAdd x2 1) ** y |-> (d%val + a%val)%val ** z |-> e).
  simpl addArgs.
  unfold someBind.
  constructor.
  searchMatch.
  simpl addrAdd.
  replace (0 + 1)%word with (1).
  reflexivity.
  womega.
  intros v'0 m'0 Hs'.
  vfUpdateStep VFE aux2 v'0; dropVfeDups VFE.
  vcSteps.
  ssimpl.
  searchMatch.
  simpl addrAdd.
```

```
replace (0 + 1)%word with (1).
reflexivity.
womega.

left.
assumption.

discriminate.

branchStep.
∃ nil.
subst c.
rewrite beq_false.
unfold resumeSpec.
unfold resumeTy.
callStep.
∃ (u |-> a ** w |-> x1 ** x |-> (addrAdd x2 1) ** y |-> d ** z |-> e).
simpl addArgs.
unfold someBind.
constructor.
searchMatch.
simpl addrAdd.
replace (0 + 1)%word with (1).
reflexivity.
womega.

intros v'0 m'0 Hs'.
vfUpdateStep VFE aux2 v'0; dropVfeDups VFE.

vcSteps.
ssimpl.
searchMatch.
simpl addrAdd.
replace (0 + 1)%word with (1).
reflexivity.
womega.

right.
subst d.
apply wordTrueVal.
simpl.
destruct eq_word_dec.
womega.
womega.

discriminate.

subst d.
subst a.
assert (impl (bopVal gtop (Vword (x3 + x0)%word) (Vword x3) = Vundef) False) as H2.
apply vundefWordBopVal with (b := gtop) (w := (x3 + x0)%word) (w' := x3).
unfold impl in H2.
apply H2.
assumption.

branchStep.
∃ nil.
subst c.
```

```
  rewrite beq_false.
  unfold resumeSpec.
  unfold resumeTy.
  callStep.
  ∃ (u |-> a ** w |-> x1 ** x |-> (addrAdd x2 1) ** y |-> d ** z |-> e).
  simpl addArgs.
  unfold someBind.
  constructor.
  searchMatch.
  simpl addrAdd.
  replace (0 + 1)%word with (1).
  reflexivity.
  womega.

  intros v'0 m'0 Hs'.
  vfUpdateStep VFE aux2 v'0; dropVfeDups VFE.

  vcSteps.
  ssimpl.
  searchMatch.
  simpl addrAdd.
  replace (0 + 1)%word with (1).
  reflexivity.
  womega.

  right.
  subst d.
  apply wordTrueVal.
  simpl.
  destruct eq_word_dec.
  womega.
  womega.

  discriminate.

  subst d.
  subst a.
  assert (impl (bopVal ltop (Vword (x3 + x0)%word) (Vword 100) = Vundef) False) as H2.
  apply vundefWordBopVal with (b := ltop) (w := (x3 + x0)%word) (w' := 100).
  unfold impl in H2.
  apply H2.
  assumption.
Qed.
```

Appendix C

Library Scheduler_Liveness

```
(*  This file contains the proof that the procedure in charge of selecting
   the next executing task will terminate as it is described in section
   3.1.2 Liveness Properties of FreeRTOS.

   The file contains two proof obligations. The first proof obligation
   demonstrates that the premise for the total correctness of the while loop
   used in selecting the next executing task is valid. In other words, only
   the inside body of the loop is proven.

   The second proof obligation shows the total correctness for the complete
   while loop construct by proving that the loop invariant is true in all
   possible execution paths and by proving that there exists a loop variant
   that decreases a finite number of times, avoiding infinite loops. *)
(*  Import Seperation Logic library *)
Require Import ProgLog.
(*  Define Global Variables *)
Definition uxTopReady := Var 0.
Definition pxReadyTaskList := Var 1.
(*  Define Local Variables *)
Definition local2 := Var 2.
Definition local3 := Var 3.
Definition local4 := Var 4.
(*  Define the body inside the while loop *)
(*  At first it was difficult to develop the proof of the while loop;
   as a sanity check and following the rule of inference for the while loop,
   we first developed the proof on the instructions inside of the body.
   After proving the body holds the postcondition if the precondition is
   given, it was easier to come up with the proof for the entire loop *)
Notation commandBody := (
  local2 ← [uxTopReady];;
  local3 ← [pxReadyTaskList] + local2;;
  local4 ← [uxTopReady];;
  [uxTopReady] ← local4 - 1;;
  ret uxTopReady
)%CM.
(*  The definition of the program, including its global and local variables *)
Notation commandDef :=
  (fdefi (uxTopReady::pxReadyTaskList::nil) (local2::local3::local4::nil) 0 commandBody).
(*  The type of the arguments passed to the program *)
(*  In this case both, uxTopReady and pxReadyTaskList are both pointers *)
Notation commandTy := (word::word::nil : tslist).
(*  The precondition that has to be met in onder for the program to hold
   the postcondition *)
```

```
(*  The precondition states that all the pointers are valid (pointing to some value),
   and that the list of tasks is not empty *)
Definition commandPre (a : word) (c : word) (b : val) (d : val) : mpred :=
  lexists index:word, lexists c1, lexists c2, lexists c3, lexists c4,
  b |-> a ** d |-> c ** (c+4)%word |-> c1 ** (c+8)%word |-> c2 ** (c+12)%word |-> c3 ** (c+16)%word
|-> c4 ** !(1 ≤ a)%word ** !(trueVal (bopVal gteop c 0)%word ∨ trueVal (bopVal gteop c1 0)%word ∨ trueVal
(bopVal gteop c2 0)%word ∨ trueVal (bopVal gteop c3 0)%word ∨ trueVal (bopVal gteop c4 0)%word) **
top.
(*  The desired final state after the program has been executed *)
(*  The postcondition states that after finishing execution, there will
   be a task to be executed next *)
Definition commandPost (a :word) (c : word) (b d _ : val) : mpred :=
  lexists v, b |-> v ** !(trueVal (bopVal gteop v 0)%word) ** top.
(*  The heap type *)
(*  Since no call to other procedures is made, the heap type is empty *)
Definition P0 : cdhpty := fun _ ⇒ None.
(*  The theorem stating that the desired final state will hold true of the program
   given that the preconditions are met *)
Lemma premiseOk : fdefOk P0 commandTy 2%nat commandPre commandDef commandPost.
Proof.
  fdefBegin. unfold commandPre, commandPost.
  intros a c b d m sp Hp vf VFE.
  simpl in VFE.
  ssimpl in Hp.
  vcSteps.
  autorewrite with ProgLog.
  unfold lex.
  ssimpl.
  ∃ (a-1)%word.
  ssimpl.
  searchMatch.
  apply wordTrueVal.
  simpl.
  destruct lte_word_dec.
  discriminate.
  womega.
Qed.
(*  Auxiliary tactics for the branch construct *)
Ltac branchStep0X :=
  match goal with
    | ⊢ context [valBoolCases ?P] ⇒
      let E := fresh "E" in
        destruct (valBoolCases P) as [[E | E] | E];
          autorewrite with valLogEq in E;
          autorewrite with valEq BlockOf in E;
          autorewrite with valEq2 valEq BlockOf in E;
            try discriminate E;
            try (elf; auto; fail)
  end.
Ltac branchStepX :=
  ∃ (None (A := spred));
    constructor; [branchStep0X |
      let s := fresh in let X := fresh in intros s X; apply X].
(*  The definition of the complete while loop *)
Notation taskSwitchBody := (
  local2 ← [uxTopReady];;
  local3 ← [pxReadyTaskList] + local2;;
  local4 ← [uxTopReady];;
  while(ebop eqop local3 0)%word (
    local4 ← [uxTopReady];;
    [uxTopReady] ← local4 - 1;;
    local2 ← [uxTopReady];;
    local3 ← [pxReadyTaskList] + local2
  );;
  ret uxTopReady
)%CM.
(*  The definition of the program, including its global and local variables *)
Notation taskSwitchDef :=
  (fdefi (uxTopReady::pxReadyTaskList::nil) (local2::local3::local4::nil) 0 taskSwitchBody).
(*  The type of the arguments passed to the program *)
(*  In this case both, uxTopReady and pxReadyTaskList are both pointers *)
Notation taskSwitchTy := (word::word::nil : tslist).
Definition listNotEmpty c c1 c2 c3 c4 :=
  trueVal (bopVal gteop c 0)%word ∨ trueVal (bopVal gteop c1 0)%word ∨ trueVal (bopVal gteop c2
0)%word ∨ trueVal (bopVal gteop c3 0)%word ∨ trueVal (bopVal gteop c4 0)%word.
(*  The precondition that has to be met in onder for the program to hold
   the postcondition *)
(*  The precondition states that all the pointers are valid (pointing to some value),
   and that the list of tasks is not empty *)
Definition taskSwitchPre (a : word) (c : word) (b : val) (d : val) : mpred :=
  lexists index:word, lexists c1, lexists c2, lexists c3, lexists c4,
  b |-> a ** d |-> c ** (c+4)%word |-> c1 ** (c+8)%word |-> c2 ** (c+12)%word |-> c3 ** (c+16)%word
|-> c4 ** !(trueVal (bopVal gteop a 0)) ** !(0 ≤ a)%word ** !(listNotEmpty c c1 c2 c3 c4 ) ** top.
(*  The desired final state after the program has been executed *)
(*  The postcondition states that after finishing execution, there will
   be a task to be executed next *)
Definition taskSwitchPost (a :word) (c : word) (b d _ : val) : mpred :=
  lexists p, b |-> p ** !(trueVal (bopVal gteop p 0)%word) ** top.
(*  The loop invariant - the condition that is true after the execution of
   all execution paths *)
Definition inv (a:word) (c:word) (b : val) (d:val) (s:cstate) :=
    ∃ lv2, ∃ lv3, ∃ lv4, (vfEqv (uxTopReady :: pxReadyTaskList :: local2 :: local3 :: local4 :: nil) ((uxTopReady, b) :: (pxReadyTaskList, d) :: (local2, lv2) :: (local3, lv3) :: (local4, lv4) :: nil) (cvfOf s) ∧
      ( lexists v:word, lexists lw3, b |-> v ** d |-> c ** !(trueVal (bopVal gteop v 0)%word) ** !(0 ≤
v)%word ** !(lv3 = Vword lw3) ** top) (cmemOf s)).
(*  An auxiliary lemma stating that the list is not empty *)
(*  The proof of this lemma has been left to a more detailed analysis
   of the initialization mechanism *)
Lemma list_not_null (lv:val) (pv:word) : lv = 0 → (0 < pv)%word.
```

```
Proof. admit. Qed.

(*  The theorem stating that the desired final state will hold true of the program
   given that the preconditions are met *)
Lemma taskSwitchOk : fdefOk P0 taskSwitchTy 2%nat taskSwitchPre taskSwitchDef taskSwitchPost.
Proof.
  fdefBegin. unfold taskSwitchPre, taskSwitchPost.
  intros a c b d m sp Hp vf VFE.
  simpl in VFE.

  ssimpl in Hp.
  vcSteps.
  ∃ (inv a c b d).
  split.

  (* Base Case *)
  unfold inv.
  ∃ a.
  ∃ (c + a)%val.
  ∃ a.
  split. intuition.
  ∃ a.
  ∃ (c + a)%word.
  ssimpl.
  searchMatch.

(*  Inductive Case *)
  clear. intros. destruct s'. unfold inv in H.
  unfold cmemOf, cspOf, cvfOf in ×.
  destruct H as [lv2 [lv3 [lv4 [VFE [v0 [w0 Hp]]]]]].
  vcSteps. branchStepX.

(*  True Branch *)
  vcSteps.
  unfold inv. ∃ (v0 - 1)%val. ∃ (c + (v0 - 1))%val. ∃ v0. split. intuition.

  ∃ (v0 - 1)%word.
  ∃ (c + (v0 - 1))%word.
  searchMatch.
  ssimpl.
  ssimpl in Hp.
  assert (0 < v0)%word.
  apply list_not_null with (lv := lv3) (pv := v0).
  assumption.
  womega.

  ssimpl in Hp.
  apply wordTrueVal.
  simpl.
  destruct lte_word_dec.
  discriminate.
  assert (0 < v0)%word.
  apply list_not_null with (lv := lv3) (pv := v0).
  assumption.
  womega.

(*  False Branch *)
```

```
  vcSteps.
  unfold lex.
  ssimpl.
  ∃ v0.
  searchMatch.
(*  Undefined Branch *)
  apply eqNullPtrVundef in E.
  assert (impl (bopVal eqop lv3 null_ptr = Vundef) False).
  ssimpl in Hp.
  replace lv3.
  apply vundefWordBopVal with (b := eqop) (w := w0) (w' := 0%word).
  unfold impl in H.
  apply H.
  assumption.
Qed.
```

Appendix D

Library Asmcost

```
(*  This file contains the cost-dynamics analysis extension added to
   Compcert to calculate the execution cost of a given program based
   on the sum of cycle counts of the generated assembly code as it
   is described in section 3.1.2 Liveness Properties of FreeRTOS.

   This file is divided into two main sections. The first one contains
   the program in charge of calculating the cost of execution. The second
   part is a list of the theorems that were needed to guarantee the
   correct behaviour of the extension added to Compcert.

   The file can be located under $COMPCERT_HOME/arm/Asmcost.v *)
(*  Import Compcert libraries *)
Require Import AST.
Require Import Asm.
Require Import List.
Require Import ListSet.
Require Import String.
Require Import EqNat.
Require Import BinPos.
Require Import Bool.
(*  Auxiliary function to determine whether a function definition
   is built-in or not. *)
Fixpoint is_builtin_function (id : ident) (lon : list (nat × string)) : bool :=
match lon with
| nil ⇒ false
| (i,s)::tl ⇒ (ifb (beq_nat (nat_of_P id) i)
(orb (prefix "__builtin_" s) (is_builtin_function id tl))
false)
end.
(*  Calculate the value of a shift operation. An immediate shift operation
   does not need an extra cycle to be computed. In any other case, an
   extra cycle is needed to compute the shift operation as described in the
   ARM documentation *)
Definition cost_shift_op (so: shift_op) : nat :=
match so with
| SOimm _ ⇒ 0
| _ ⇒ 1
end.
(*  Calculate the cost of a given assembly instruction *)
Definition cost_instr (i : Asm.instruction) (lon : list (nat × string)) : nat :=
match i with
| Padd _ _ so ⇒ 1 + (cost_shift_op so) (* *r integer addition *)
| Pand _ _ so ⇒ 1 + (cost_shift_op so) (* *r bitwise and *)
| Pb _ ⇒ 3 (* *r branch to label *)
| Pbc _ _ ⇒ 3 (* *r conditional branch to label *)
| Pbsymb id ⇒ (if (is_builtin_function id lon) then 1 else 3) (* *r branch to symbol *)
| Pbreg _ ⇒ 3 (* *r computed branch *) (*  PC destination *)
| Pblsymb id ⇒ (if (is_builtin_function id lon) then 12 else 3) (* *r branch and link to symbol *)
| Pblreg _ ⇒ 4 (* *r computed branch and link *)
| Pbic _ _ so ⇒ 1 + (cost_shift_op so) (* *r bitwise bit-clear *)
| Pcmp _ so ⇒ 1 + (cost_shift_op so) (* *r integer comparison *)
| Peor _ _ so ⇒ 1 + (cost_shift_op so) (* *r bitwise exclusive or *)
| Pldr _ _ _ ⇒ 3 (* *r int32 load *)
| Pldrb _ _ _ ⇒ 3 (* *r unsigned int8 load *)
| Pldrh _ _ _ ⇒ 3 (* *r unsigned int16 load *)
| Pldrsb _ _ _ ⇒ 3 (* *r signed int8 load *)
| Pldrsh _ _ _ ⇒ 3 (* *r unsigned int16 load *)
| Pmov _ so ⇒ 1 + (cost_shift_op so) (* *r integer move *)
| Pmovc _ _ so ⇒ 1 + (cost_shift_op so)(* *r integer conditional move *)
| Pmul _ _ _ ⇒ 5 (* *r integer multiplication *)
| Pmvn _ so ⇒ 1 + (cost_shift_op so) (* *r integer complement *)
| Porr _ _ so ⇒ 1 + (cost_shift_op so) (* *r bitwise or *)
| Prsb _ _ so ⇒ 1 + (cost_shift_op so) (* *r integer reverse subtraction *)
| Pstr _ _ _ ⇒ 2 (* *r int32 store *)
| Pstrb _ _ _ ⇒ 2 (* *r int8 store *)
| Pstrh _ _ _ ⇒ 2 (* *r int16 store *)
| Psdiv _ _ _ ⇒ 10 (* *r signed division *)
| Psub _ _ so ⇒ 1 + (cost_shift_op so) (* *r integer subtraction *)
| Pudiv _ _ _ ⇒ 10 (* *r unsigned division *)
| Pabsd _ _ ⇒ 4 (* *r float absolute value *)
| Padfd _ _ _ ⇒ 10 (* *r float addition *)
| Pcmf _ _ ⇒ 7 (* *r float comparison *)
| Pdvfd _ _ _ ⇒ 65 (* *r float division *)
| Pfixz _ _ ⇒ 8 (* *r float to signed int *)
| Pfltd _ _ ⇒ 9 (* *r signed int to float *)
| Pldfd _ _ _ ⇒ 8 (* *r float64 load *)
| Pldfs _ _ _ ⇒ 8 (* *r float32 load *)
| Plifd _ _ ⇒ 8 (* *r load float constant *)
| Pmnfd _ _ ⇒ 8 (* *r float opposite *)
| Pmvfd _ _ ⇒ 8 (* *r float move *)
| Pmvfs _ _ ⇒ 8 (* *r float move single precision *)
| Pmufd _ _ _ ⇒ 8 (* *r float multiplication *)
| Pstfd _ _ _ ⇒ 8 (* *r float64 store *)
| Pstfs _ _ _ ⇒ 8 (* *r float32 store *)
| Psufd _ _ _ ⇒ 8 (* *r float subtraction *)
(*  Pseudo-instructions *)
| Pallocframe _ _ _ ⇒ 7 (* *r allocate new stack frame *)
| Pfreeframe _ _ _ ⇒ 3 (* *r deallocate stack frame and restore previous frame *)
| Plabel _ ⇒ 0 (* *r define a code label *)
| Ploadsymbol _ _ _ ⇒ 3 (* *r load the address of a symbol *)
| Pbtbl _ _ ⇒ 5 + 1 (* *r N-way branch through a jump table *)
| Pbuiltin _ _ _ ⇒ 5 (* *r built-in *)
end.

(*  The cost of the code inside a function definition is the sum of the cost
   of all individual instructions that define the function definition. *)
```

```
Definition cost_code (il : list Asm.instruction) (lon : list (nat × string)) : nat :=
List.fold_right plus 0 (List.map (fun par ⇒ (cost_instr par lon)) il).

(*  The cost of a single function definition. The cost of external function
   definitions cannot be calculated since the internal instructions are
   not known *)
Definition cost_fundef (f : Asm.fundef) (lon : list (nat × string)) : nat :=
match f with
| External _ ⇒ 0
| Internal c ⇒ cost_code c lon
end.

(*  The cost of a list of function definitions is the sum of the cost of
   the first function definition plus the cost of the rest of function
   definitions *)
Fixpoint cost_list_fd (fdl : list (ident × fundef)) (lon : list (nat × string)) : nat :=
match fdl with
| nil ⇒ 0
| cons (i,fd) rem ⇒ cost_fundef fd lon + cost_list_fd rem lon
end.

(*  The cost of execution of a program is the cost of execution of the
   list of function definitions *)
Definition cost_program (p : Asm.program) (lon : list (nat × string)) : nat :=
cost_list_fd (prog_funct p) lon.

(*  The following is the definition of the extension to
   the AST.v file under $compcert_arm/common along with
   some remarks (theorems) that will facilitate the proof of correctness *)

Section MAP_PARTIAL_ARG.

Variable A B C: Type.
Variable prefix_errmsg: A → errmsg.
Variable f: B → list (nat × string) → res C.
Variable arg : list (nat × string).

Fixpoint map_partial_arg (l: list (A × B)) : res (list (A × C)) :=
  match l with
  | nil ⇒ OK nil
  | (a, b) :: rem ⇒
      match f b arg with
      | Error msg ⇒ Error (prefix_errmsg a ++ msg)%list
      | OK c ⇒
          do rem' ← map_partial_arg rem;
          OK ((a, c) :: rem')
      end
  end.

Remark In_map_partial_arg:
  ∀ l l' a c,
  map_partial_arg l = OK l' →
  In (a, c) l' →
  ∃ b, In (a, b) l ∧ f b arg = OK c.
Proof.
  induction l; simpl.
  intros. inv H. elim H0.
  intros until c. destruct a as [a1 b1].
  caseEq (f b1 arg); try congruence.
  intro c1; intros. monadInv H0.
  elim H1; intro. inv H0. ∃ b1; auto.
  exploit IHl; eauto. intros [b [P Q]]. ∃ b; auto.
Qed.

Remark map_partial_arg_forall2:
  ∀ l l',
  map_partial_arg l = OK l' →
  list_forall2
    (fun (a_b: A × B) (a_c: A × C) ⇒
       fst a_b = fst a_c ∧ f (snd a_b) arg = OK (snd a_c))
    l l'.
Proof.
  induction l; simpl.
  intros. inv H. constructor.
  intro l'. destruct a as [a b].
  caseEq (f b arg). 2: congruence. intro c; intros. monadInv H0.
  constructor. simpl. auto. auto.
Qed.

End MAP_PARTIAL_ARG.

Remark map_partial_arg_total:
  ∀ (A B C: Type) (prefix: A → errmsg) (f: B → list (nat × string) → C) (myl: list (nat × string)) (l:
list (A × B)),
  map_partial_arg prefix (fun b arg ⇒ OK (f b arg)) myl l =
  OK (List.map (fun a_b ⇒ (fst a_b, f (snd a_b) myl)) l).
Proof.
  induction l; simpl.
  auto.
  destruct a as [a1 b1]. rewrite IHl. reflexivity.
Qed.

Remark map_partial_arg_identity:
  ∀ (A B: Type) (prefix: A → errmsg) (myl: list (nat × string)) (l: list (A × B)),
  map_partial_arg prefix (fun b arg ⇒ OK b) myl l = OK l.
Proof.
  induction l; simpl.
  auto.
  destruct a as [a1 b1]. rewrite IHl. reflexivity.
Qed.

(*  PROOF OF CORRECTNESS *)
```

The following is a variant of *transform_program_partial* where a extra argument of function names is passed to the transformation `Section` *TRANSF_PARTIAL_PROGRAM_ARG*.

```
Variable A B V: Type.
Variable transf_partial: A → list (nat × string) → res B.

Definition transform_partial_program_arg (p: program A V) (myl: list (nat × string)) : res (program B
V) :=
  do fl ← map_partial_arg prefix_name transf_partial myl p.(prog_funct);
  OK (mkprogram fl p.(prog_main) p.(prog_vars)).

(*  A program consist of
   - The list of function definitions
   - The main function that serves as the entry point
   - The list of variables

   The proof consists on the correct translation of each part of the program *)
Lemma transform_partial_program_arg_function:
  ∀ p l tp i tf,
  transform_partial_program_arg p l = OK tp →
  In (i, tf) tp.(prog_funct) →
  ∃ f, In (i, f) p.(prog_funct) ∧ transf_partial f l = OK tf.
Proof.
  intros. monadInv H. simpl in H0.
  eapply In_map_partial_arg; eauto.
Qed.
Lemma transform_partial_program_arg_main:
  ∀ p l tp,
  transform_partial_program_arg p l = OK tp →
  tp.(prog_main) = p.(prog_main).
Proof.
  intros. monadInv H. reflexivity.
Qed.
Lemma transform_partial_program_args_vars:
  ∀ p l tp,
  transform_partial_program_arg p l = OK tp →
  tp.(prog_vars) = p.(prog_vars).
Proof.
  intros. monadInv H. reflexivity.
Qed.
End TRANSF_PARTIAL_PROGRAM_ARG.
(*  Adding an extension to Compcert required to change some parts
   of the code. While the changes were not sbstantial, they did require
   to make slight modifications to some of the theorems, none of them
   affecting the preservation of corrcetness.

   The following is a sample of some of the theorems that required changes.
   In general, an additional argument with the names of the function definitions
   was passed to the phases of the compiler even though such argument was
   ignored for all of them except the cost of execution. *)
Theorem transf_rtl_program_correct:
  ∀ p l tp beh,
  transf_rtl_program p l = OK tp → (*  The list of function names "l" is passes as parameter *)

  not_wrong beh →
  RTL.exec_program p beh →
  Asm.exec_program tp beh.
Proof.
  intros. unfold transf_rtl_program, transf_rtl_fundef in H.
  repeat TransfProgInv.
  repeat rewrite transform_program_print_identity in ×. subst.
  exploit transform_partial_program_identity; eauto. intro EQ. subst.
  generalize Alloctyping.program_typing_preserved Tunnelingtyping.program_typing_preserved
             Linearizetyping.program_typing_preserved Reloadtyping.program_typing_preserved
             Stackingtyping.program_typing_preserved; intros.
  eapply Asmgenproof.transf_program_correct; eauto 6.
  eapply Machabstr2concr.exec_program_equiv; eauto 6.
  eapply Stackingproof.transf_program_correct; eauto.
  eapply Reloadproof.transf_program_correct; eauto.
  eapply Linearizeproof.transf_program_correct; eauto.
  eapply Tunnelingproof.transf_program_correct; eauto.
  eapply Allocproof.transf_program_correct; eauto.
  eapply CSEproof.transf_program_correct; eauto.
  eapply Constpropproof.transf_program_correct; eauto.
  eapply CastOptimproof.transf_program_correct; eauto.
  eapply Tailcallproof.transf_program_correct; eauto.
Qed.

Theorem transf_cminor_program_correct:
  ∀ p l tp beh,
  transf_cminor_program p l = OK tp → (*  The list of function names "l" is passes as parameter *)

  not_wrong beh →
  Cminor.exec_program p beh →
  Asm.exec_program tp beh.
Proof.
  intros. unfold transf_cminor_program, transf_cminorsel_fundef in H.
  simpl in H. repeat TransfProgInv.
  eapply transf_rtl_program_correct with (l := l); eauto.
  eapply RTLgenproof.transf_program_correct; eauto.
  eapply Selectionproof.transf_program_correct; eauto.
  rewrite print_identity. auto.
Qed.

Theorem transf_c_program_correct:
  ∀ p l tp beh,
  transf_c_program p l = OK tp → (*  The list of function names "l" is passes as parameter *)

  not_wrong beh →
  Cstrategy.exec_program p beh →
  Asm.exec_program tp beh.
Proof.
  intros until beh; unfold transf_c_program; simpl.
  rewrite print_identity.
  caseEq (SimplExpr.transl_program p); simpl; try congruence; intros p0 EQ0.
  rewrite print_identity.
  caseEq (Cshmgen.transl_program p0); simpl; try congruence; intros p1 EQ1.
  caseEq (Cminorgen.transl_program p1); simpl; try congruence; intros p2 EQ2.
  intros EQ3 NOTW SEM.
  eapply transf_cminor_program_correct; eauto.
  eapply Cminorgenproof.transl_program_correct; eauto.
  eapply Cshmgenproof.transl_program_correct; eauto.
  eapply SimplExprproof.transl_program_correct; eauto.
Qed.
```