FORMAL VERIFICATION OF DEVICE DRIVERS IN EMBEDDED SYSTEMS

by

Jianjun Duan

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

December 2013

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Jianjun Duan**

has been approved by the following supervisory committee members:

John Regehr	, Chair	**Jul 8th, 2013** Date Approved
Matthew Flatt	, Member	**July 8th, 2013** Date Approved
Ganesh Gopalakrishnan	, Member	**July 8th, 2013** Date Approved
Michael Norrish	, Member	**July 15th, 2013** Date Approved
Konrad Slind	, Member	**July 18th, 2013** Date Approved

and by **Alan Davis** , Chair/Dean of

the Department/College/School of **Computing**

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

Embedded systems are often deployed in a variety of mission-critical fields, such as car control systems, the artificial pace maker, and the Mars rover. There is usually significant monetary value or human safety associated with such systems. It is thus desirable to prove that they work as intended or at least do not behave in a harmful way.

There has been considerable effort to prove the correctness of embedded systems. However, most of this effort is based on the assumption that embedded systems do not have any peripheral devices and interrupt handling. This is too idealistic because embedded systems typically depend on some peripheral devices to provide their functionality, and in most cases these peripheral devices interact with the processor core through interrupts so that the system can support multiple devices in a real time fashion.

My research, which focuses on constrained embedded systems, provides a framework for verifying realistic device driver software at the machine code level. The research has two parts.

In the first part of my research, I created an abstract device model that can be plugged into an existing formal semantics for an instruction set architecture. Then I instantiated the abstract model with a model for the serial port for a real embedded processor, and plugged it into the ARM6 instruction set architecture (ISA) model from the University of Cambridge, and verified full correctness of a polling-based open source driver for the serial port.

In the second part, I expanded the abstract device model and the serial port model to support interrupts, modified the latest ARMv7 model from the University of Cambridge to be compatible with the abstract device model, and extended the Hoare logic from the University of Cambridge to support hardware interrupt handling. Using this extended tool chain, I verified full correctness of an interrupt-driven open source driver for the serial port.

To the best of my knowledge, this is the first full correctness verification of an interrupt-driven device driver. It is also the first time a device driver with inherent timing constraints has been fully verified. Besides the proof of full correctness for realistic serial port drivers, this research produced an abstract device model, a formal specification of the circular buffer at assembly level, a formal specification for the serial port, a formal ARM system-on-chip (SoC) model which can be extended by plugging in device models, and the inference rules to reason about interrupt-driven programs.

For family and friends

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First of all, I want to thank my advisor, Professor John Regehr. He guided me into the area of applying formal methods on embedded systems, and helped me to develop the dissertation topic. Without his guidance and support, this dissertation would not be possible.

I am also very grateful to my former advisor, Dr. Konrad Slind. He taught me a lot about the formal methods, especially the theorem prover HOL, which is the major tool I used in my dissertation work.

I also thank my committee members: Professor Ganesh Gopalakrishnan, Professor Matthew Flatt, and Dr. Michael Norrish. I learned a lot about formal methods, functional programming and HOL from them. Discussion with them helps me expand the horizon of my research.

I also had inspiring discussions about the ARM model inside HOL and research ideas with Dr. Anthony Fox and Dr. Magnus Myreen from the University of Cambridge. I thank them for their contribution to my research.

Fellow students and friends, Xiaofang Chen, Yang Chen, Guodong Li, Peng Li, Xuejun Yang, Yu Yang and Lu Zhao, also helped me a lot in these years both in academics and daily life. I thank them for their help.

I also thank Karen Feinauer and Ann Carlstrom. They handled all my paperwork and helped me a lot on technical problems outside my research. It was great to have them in the front office in these years.

CHAPTER 1

INTRODUCTION

Embedded systems are often deployed in fields where there is significant risk of human safety or property. Thus it is desirable to guarantee that they work as intended or at least do not behave harmfully. Some examples are car control systems, artificial pace makers, and the Mars rover.

Generally speaking, formal verification can be used to provide rigorous guarantees. However, the vast majority of verification work on embedded systems ignores devices and device drivers. This is problematic because an embedded system is tailored around a set of devices to perform a dedicated task.

The tailoring is done at both hardware and software levels. At the hardware level, only necessary peripheral devices are built into the system without much flexibility to expand. Some common peripheral devices include the serial port, analog-digital converter (ADC), and timer. At the software level, the system can be viewed as little more than necessary device drivers and a simple control loop. Device drivers are often specialized.

In this dissertation I will address this deficiency in current research of formal verification of embedded systems using a theorem prover. My goal is to prove the correctness of realistic drivers on embedded systems with peripheral devices working in interrupt mode. I will develop a framework. The effectiveness of the framework will be demonstrated by some realistic examples.

1.1 Thesis Statement

Formal proof about functional correctness of realistic device drivers on embedded systems with memory mapped devices working in interrupt mode can be done using theorem proving tools.

Previous work in low-level machine models and machine-level program logic using theorem proving tools provides a good foundation. My idea is to expand existing machine models and program logic to support hardware devices and interrupts.

First, I introduce a device model which can be combined with an instruction set architecture (ISA) model to verify device drivers. Then it is extended to support interrupts. Finally, I add program logic support for reasoning software taking advantage of interrupts.

It will be implemented in HOL [39] on top of the ARM ISA model [33, 60]. The method is applied to verify functional correctness of an open source driver for the serial port, which is used to receive and transmit data.

1.2 Challenges

There are substantial challenges for this dissertation work. To verify an interrupt-driven system, it is better to work at the assembly[1] level. This, however, exposes all the low-level details. Parallelism associated with interrupts is complicated because of side effects in device register access. Realistic drivers and ISRs often have inherent timing constraints. Rigorous proof often comes at the cost of laborious effort.

1.2.1 Low-Level Nature

The programmability of a system at register level allows a wide range of flexibility in software, which hinders the portability of some generic methodologies. One example is the variety of interrupt handling schemes that can be implemented on the same hardware platform. Another one is the optimizations of a program at assembly level.

Access of device registers has side effects. One has to have an accurate understanding of them at the register level to reason about device drivers. Such side effects make the concurrency between interrupt handlers and the main program more complicated than the one between threads.

1.2.2 Parallelism and Concurrency

As shown in Figure 1.1, a device runs in parallel with the processor core. Its state changes by itself (with possible input from the environment) or as the side effect of

[1] I use assembly and machine language interchangeably in verification context.

commands from the processor core. The interaction between them can be two-way.

The processor core sends a command to a device by reading from or writing to some particular memory address which is mapped to a register on the device. The devices can force the processor core to execute some code, called interrupt handlers, by interrupting its normal execution of the main program through interrupts, which are triggered by some hardware events. The device driver will package such interactions, and provide a high-level application program interface (API).

Now, even for a system with a single thread, a device driver (execution of the processor core) is a parallel entity to the devices that it manages, and the interrupt handlers run concurrently with the main program. This makes testing not a convincing approach to provide trustworthy guarantees about the functionality and safety of the system, because complete coverage is not possible.

1.2.3 Timing Properties

Often interesting properties of an embedded system are associated with time. For example, it is often assumed that interrupts are handled in a timely manner. Only then can the function of an interrupt service routine (ISR)[2] be guaranteed. Another example is that there are always restrictions on the speed of devices, such as data throughput and the maximum frequency of interrupt requests.

In temporal logic [19, 24, 49, 73], time is modeled as one time line or a tree structure with multiple possible future branches. Here, the notion of time describes the rate at which an event can occur. Formalization of embedded systems needs to account for it.

1.2.4 Mechanical Proof

The ideal guarantee I am after is a mathematical proof. However, given the challenges, it is not plausible by a paper-and-pencil proof as done in an algorithm class. A mechanical proof is necessary.

To verify nontrivial properties of a piece of realistic software on a realistic embedded system, one needs to formalize the system with plenty of low-level details, reason about concurrency abstractly at a high level, and automate as much as possible.

[2]I use ISR and interrupt handlers interchangeably in this dissertation.

A theorem prover is necessary for achieving these tasks, especially to ensure the soundness of proofs.

1.3 Background

This dissertation work involves formalizing an abstract device model which can used in verification of different devices and their drivers, the operational semantics of ARM assembly language on a machine with devices, program logic support for high-level proof, and formalization using HOL theorem prover. Here is a brief overview of these subjects.

1.3.1 Devices

Formal semantics of a language are usually defined on an abstract machine. In most cases, the state of the machine consists of memory and registers. However, in practice, often extensions are made to languages so that the semantics are defined on machines whose states include more than memory and registers.

One such extension is memory-mapped I/O. It introduces side effects to memory access when the address is mapped to a device register. This means that to prove properties about a program which accesses memory-mapped I/O, formal semantics need be developed for the language on an abstract machine with devices. Side effects need be addressed.

Such formal semantics should be built for assembly languages. Doing so means compilers are not in the Trusted Computing Base (TCB) any more. There is plenty of evidence suggesting that compilers cannot be trusted [101]. Assembly language is still used in mission critical embedded systems for reasons such as performance and resource constraints. Also, for device operations as well as interrupt handling, high-level languages do not provide fine enough granularity in terms of temporal resolution.

1.3.2 Operational Semantics

On the other hand, we can just use an operational semantics of a language to prove the properties of a program, as pointed out by Moore [57]. In operational semantics, the meaning of a program is seen as a sequence of transition on the state which the program lives on. Such transition is described by a step function: $s_t = \texttt{step}\ (s_{t-1})$. When the operational semantics of a programming language are formalized in an interactive theorem prover, one can try to prove almost any goal conceivable about a program. In most cases however, it is not efficient, since lots of work will be repeated in a proof.

To improve the performance, a Hoare logic [31, 42] can be built on top of the formal operational semantics. Program logic provides high-level inference rules. In tricky situations when high-level inference rules fail, one can always attempt the proof at the level of operation semantics.

1.3.3 Hoare Logic

When proving some property about a program, the program must be treated as a mathematical object. Hoare logic provides a way to do so. A program is built from statements following grammar rules. In Hoare logic, an axiom is used to describe the precondition and postcondition of a statement in the form of $\{P\}\ C\ \{Q\}$. This is a Hoare triple. Here, C is the statement, P is the precondition assertion, and Q is the postcondition assertion. For each grammar rule, corresponding inference rules are given.

The property of a program is also specified in terms of a Hoare triple. There are two ways to prove the Hoare triple for a program. One is the top-down approach, in which the Hoare triple for a program is broken down into subgoals about the statements of the program using inference rules. This process is called verification condition generation. Another is the bottom-up approach, in which inference rules are used to compose the axioms of individual statements together to form the final theorem about the program.

Using Hoare logic to prove properties about a realistic program in a realistic programming language is not easy. One problem is efficiency. Burstall [15] realized that for some data structures like list, the naive way of specifying memory properties would not scale. Separation logic [80] is developed to address such issues. In separation logic reasoning is done locally using the frame rule: $\{P\}\ C\ \{Q\} \Rightarrow \{P\ *\ P^{'}\}\ C\ \{Q\ *\ Q^{'}\}$. Here, * is separation conjunction, $x\ *\ y$ means x and y hold on two separated sets of resources. Further down the line concurrent separation logic [13] is developed to reason about concurrency.

Another problem is how to support language features in terms of inference rules. Most commonly used languages present their own unique challenges. For example, a Hoare logic for C needs to address the flexible memory model, like in L4 verification project [95]. For assembly languages, rather complex control flow has to be supported [93].

1.3.4 HOL

HOL is an interactive theorem prover for higher order logic. It is built using the LCF approach [40], which gives it soundness guarantee on the theorems proved using it. A lot of work has been done to formalize the ARM ISA in HOL by Fox et al. [33, 37] at the University of Cambridge. It is low level and faithful to the ARM standard as demonstrated by the verification of the ARM6 microarchitecture.

The Hoare logic [60] built on top of it supports separation logic style reasoning. For example, ARM instruction STRB r2, [r3] stores the least significant byte in register r2 at the memory location pointed to by the pointer in register r3. The Hoare triple for it is:

$$
\begin{aligned}
&(\mathsf{aPC}\ p * \mathsf{aR}\ 3\ a * \mathsf{aR}\ \ 2\ b * \mathsf{aBYTE\_MEMORY}\ \mathit{df}\,f * \mathsf{cond}(a \in \mathit{df}))\\
&\ \{\ (p, \mathsf{0xE5C32000})\ \}\\
&(\mathsf{aPC}\ (p\ +4) * \mathsf{aR}\ 3\ a * \mathsf{aR}\ 2\ b * \mathsf{aBYTE\_MEMORY}\ \mathit{df}\ ((a \mapsto b{\uparrow}^{8}_{32})\ f)).
\end{aligned}
$$

It states this at the machine level: If the program counter holds value p ($\mathsf{aPC}\ p$), register 2 holds value b ($\mathsf{aR}\ 2\ b$), register 3 holds value a ($\mathsf{aR}\ 3\ a$), and instruction 0xE5C32000 (the binary encoding of STRB r2, [r3]) resides at the address p of the memory (p, 0xE5C32000), and that a is a valid memory address ($\mathsf{cond}\ (a \in \mathit{df})$), then after the execution of the processor the program counter will increase by 4, and the memory ($\mathsf{aBYTE\_MEMORY}\ \mathit{df}\,f$) will be updated such that the cell at address a now holds value b. $b\uparrow^{8}_{32}$ is the least significant byte of b. for an ARM assembly program, if theorems like this for all its instructions are available, they can be composed to form the specification of the program. My work is built upon this ARM ISA model. I extended the ARM ISA model to support device access, and extended the Hoare logic to support interrupt handling.

1.4 Outline

I present a progressive work in two parts in this dissertation. The flow of the technical work including ideas and theorems is shown in Figure 1.2. An arrow from A to B indicates the idea or theorems in A is expanded or applied in B. Chapters 2 and 3 form the first part, which is about verifying a polling based device driver. An abstract device model for using with an ISA model is presented in Chapter 2, which forms the foundation of my work. In Chapter 3, I present the serial port model without supporting interrupts, and prove the functional correctness of a polling-based serial port driver.

Chapters 4, 5 and 6 form the second part. In this part I present my approach to verify interrupt-driven device drivers and demonstrate it by verifying an interrupt-driven serial port driver. In Chapter 4, I overview ARM related research work done at the University of Cambridge, and develop the semantics for the ARM SoC model, as well as the support in program logic. The correctness proof for the ISR of the serial port as a standalone program is presented in Chapter 5. In Chapter 6, I present the correctness proof of an interrupt-driven serial port driver.

I summarize the work in Chapter 7. Future work is proposed in Chapter 8, and related work is discussed in Chapter 9.

1.5 Notation

Here, I introduce some of the notations used in this dissertation.

1. Almost all the constant numbers are actually words of fixed width, except the parameters I use to count time elapse.
2. I used different fonts to differentiate a variable and a function or keyword. $\mathsf{defined}$ is a keyword. $\mathsf{function}$ is a function name. var is a variable.
3. Every variable is a free variable at the top level[3] unless it is explicitly qualified.
4. Every theorem starts with $\vdash$. Small theorems are presented in equations, while large ones are in figures.

[3]It is equivalent to a universally qualified variable in HOL as far as the meaning of theorems is concerned.

5. Underlined terms such as $\underline{\textit{placeholder}}$ are not ordinary variables. They are placeholders for constant and complex formulas, and used to make formula and theorems more readable.
6. For some processor registers and device registers, it is common to refer to the individual bits of them. I use the capitalized bit name as subscript to denote a bit in a register. For example, $\mathsf{PSR}_{\mathrm{I}}$ means the interrupt mask bit in a processor status register (PSR), $\mathsf{IER}_{\mathrm{RLS}}$ denotes the receiving line status (RLS) bit in the Interrupt Enable Register (IER) of the serial port.
7. Definition is indicated by $\triangleq$.

Memory is modeled as a map from 32-bit word to byte (8-bit word). Little-endianness is followed. Following symbols are used to denote some operations on memory and words:

1. $f \nearrow^{\mathrm{n}}_{\mathrm{m}} \ addr$ means reading m bytes from addresses starting from $addr$ first, then concatenating them into a $m \times 8$-bit word, and finally converting it to a n-bit word by zero-extending it or discarding the extra more significant bits.
2. $\overleftarrow{addr}^{\mathrm{n}}$ returns the set $\{addr,\ addr+1, \cdots, addr+n-1\}$. It is used to return the set of addresses starting from the given address in a multibyte memory access.
3. $x{\upharpoonleft}^{\mathrm{n}}_{\mathrm{m}}$ converts a m-bit word x to a n-bit word by zero-extending it or discarding the extra more significant bits.
4. $+_x$ denotes modular word addition with regard to x.
5. Subscript $+$ in comparison operators such as $\leq_+$ means unsigned operation.

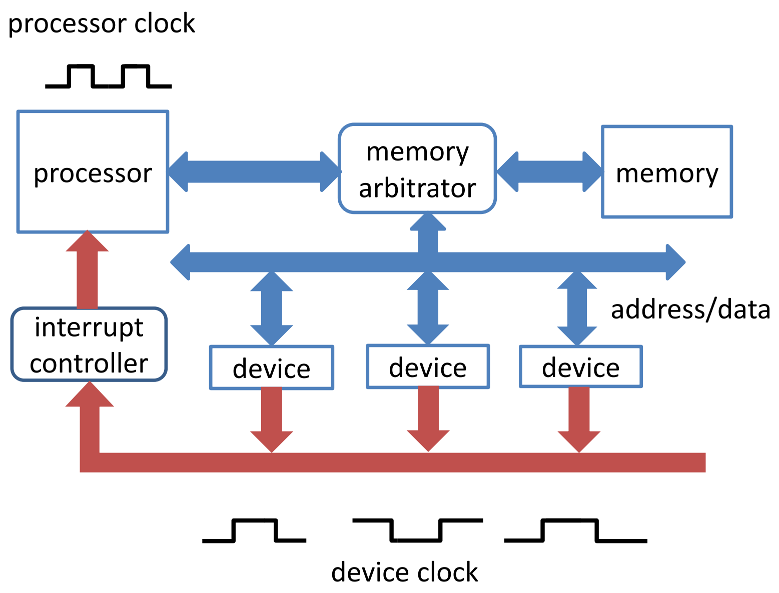

Figure 1.1: System with devices

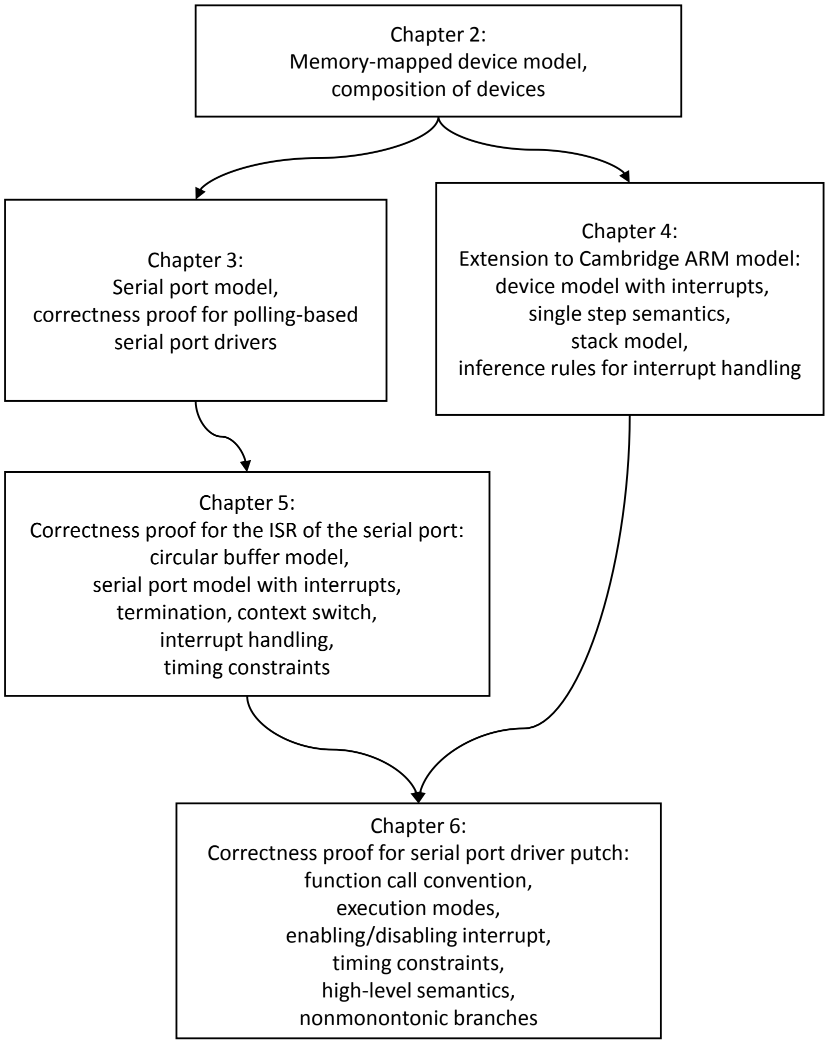
Chapter 2:
Memory-mapped device model,
composition of devices
Chapter 3:
Serial port model,
correctness proof for polling-based
serial port drivers
Chapter 4:
Extension to Cambridge ARM model:
device model with interrupts,
single step semantics,
stack model,
inference rules for interrupt handling
Chapter 5:
Correctness proof for the ISR of the serial port:
circular buffer model,
serial port model with interrupts,
termination, context switch,
interrupt handling,
timing constraints
Chapter 6:
Correctness proof for serial port driver putch:
function call convention,
execution modes,
enabling/disabling interrupt,
timing constraints,
high-level semantics,
nonmonontonic branches

Figure 1.2: The flow of technical work

CHAPTER 2

DEVICE MODEL

In this chapter I present an abstract device model which is to be used with an ISA model. The target ISA model is the ARMv4 model [33] by Fox.

2.1 Overview

Fox's ARM model includes banked registers including special purpose registers, exceptions, coprocessors, and a data bus. A system model with no coprocessors and no interrupt handling is built by extending the core model with memory through the data bus. Its next-state operation is at the instruction level; it takes a system state as the input and returns the next state as the result of current instruction execution. There are tools to automatically prove theorems about the semantics of individual concrete ARMv4 instructions.

2.1.1 System Model

An embedded system is modeled using this record:

$$<|\mathsf{next} : \mathsf{state} \rightarrow \mathsf{state}; \mathsf{is\_undefined} : \mathsf{state} \rightarrow \mathsf{bool}|>. \tag{2.1}$$

$\rightarrow$ is used to represent function types. state is the type of the state of the system. next is used to encode the transition of the system, which includes fetching and parsing an instruction, fetching data, computing, updating registers, memory, and the state of devices. The is_undefined predicate tests if a state is erroneous. The system enters an erroneous state when the processor core encounters an exceptional condition[1], when the processor core accesses the memory addresses which are mapped to some device that is not present in the system, or when a device-specific error is encountered, i.e.,

[1]I do not consider the handling of interrupts or processor exceptions here.

reading a device register that is in an indeterminate state or writing to a read-only device register.

It is required that:

$$\mathsf{is\_undefined}\ s \Rightarrow \mathsf{is\_undefined}\ (\mathsf{next}\ s).$$

That is, an erroneous state is sticky and we are not concerned with the system's subsequent behavior. One of my goals will be to prove that device drivers cannot put the system into an erroneous state.

A function step describes the effect of consecutive application of next:

$$\mathsf{step}\ n\ s \triangleq \mathsf{if}\ n = 0\ \mathsf{then}\ s\ \mathsf{else}\ \mathsf{next}\ (\mathsf{step}\ (n-1)\ s).$$

For step to describe a running system, memory values including both instructions and data must be part of the system state. But that is not enough. For example, the processor can use values obtained by a sensor device from the environment to change the memory content. For cases like this, it is required that the state of the related device contain input streams, which need to contain the information from the future.

2.1.2 Assertion Formula

I use the following construct to assert the properties of a state for the system:

$$\begin{aligned}
&\mathsf{sys\_pred}\ P\ I\ Q \triangleq \\
&\forall s.\ P\ s \wedge \neg\mathsf{is\_undefined}\ s \Rightarrow \\
&\quad \exists t.\ Q\ (\mathsf{step}\ t\ s) \wedge \\
&\qquad \forall n.\ n \leq t \Rightarrow I\ (s, \mathsf{step}\ n\ s) \wedge \neg\mathsf{sys\_undefined}\ (\mathsf{step}\ n\ s\ ).
\end{aligned} \tag{2.2}$$

This is a shallow embedding of Hoare logic with P, I, and Q as the predicates of precondition, global invariant, and postcondition with type $\mathsf{state} \to \mathsf{bool}$. Note that this is about complete correctness.

To use this construct to describe the properties of a program, the program must be specified in terms of the current program counter and instruction memory. The value of the program counter and the instruction memory should be specified as part of P. I represent the program as a set of pairs of an instruction and its address. I use

$\mathsf{code}\ p\ s$ to indicate a program p is part of the memory in a system state s.

In most cases, the part of memory which holds the program should be left unchanged at every moment. That should be part of I.

To prove a claim in the form of Equation 2.2, an instance of t needs to be found first. Then induction on n is used. I use this method to prove the full correctness of a serial port driver working in the polling mode.

2.2 Memory-Mapped I/O

Embedded systems mostly have peripheral devices built in. The mechanism is memory-mapped I/O. Here, I introduce an abstract device model for memory-mapped I/O. It is intended to work with an ISA model.

2.2.1 Abstract Device Model

A peripheral device runs in parallel with the processor core. Its state can change with or without interacting with the core or with the external world. The core interacts with devices using memory-mapped I/O: a collection of dedicated registers that are mapped into the processor's address space. From the perspective of the core, these registers are accessed like memory locations, though of course device registers do not in general contain the last value written to them, and both reads and writes may have side effects.

Based on this observation, I design an abstract type to represent a generic memory-mapped peripheral device:

$$\begin{aligned}&\mathsf{<|\,mapped} : \mathsf{addr} \rightarrow \mathsf{bool};\\ &\quad \mathsf{mapped\_read} : \mathsf{addr} \rightarrow \tau \rightarrow \mathsf{word} * \ \mathsf{bool} * \tau;\\ &\quad \mathsf{mapped\_write} : \mathsf{addr} \rightarrow \mathsf{data} \rightarrow \tau \rightarrow \mathsf{bool} * \tau;\\ &\quad \mathsf{transit} : \tau \rightarrow \tau; \mathsf{wellform} : \tau \rightarrow \mathsf{bool\,|>}. \end{aligned} \tag{2.3}$$

Here, addr and word are types for memory addresses and data. τ is the type for the state of the device, which varies depending on the individual device. $*$ is used to construct a tuple. mapped describes if an address is mapped to this device. $\mathsf{mapped\_read}$ and $\mathsf{mapped\_write}$ describe the effect of read and write commands from the processor core. Possible side effect on the device states is captured by τ in input and return types. The flag with bool type indicates if an error occurs during the memory-mapped access of the device registers. $\mathsf{transit}$ describes the autonomous transition of the device itself without the command from the processor core. $\mathsf{wellform}$ tells if a state of the device is wellformed. I use τ dev to refer to the type of the device as in Equation 2.3.

A concrete device such as a serial port is modeled as an instance of this abstract model. τ is instantiated with a concrete type, and all the members are assigned functions which model the concrete device.

2.2.2 Device Pool

Device models can be repeatedly combined as long as they fail to share mapped registers (real devices have this property, generally). The operation is defined in Figure 2.1. Here, FST and SND return the first and second member of a pair type, respectively. With $\mathsf{comb\_dev}$ I can construct and reason about an embedded system with multiple devices modularly.

2.3 Extended System Model

An embedded SoC is a processor core plus a collection of peripherals. I start with a processor core that is extended with a null device whose mapped function returns false for all addresses. I can then build a realistic SoC as shown in Figure 2.2 by adding more devices on top of this bare one, using the operations in Figure 2.1.

The state of a system with devices is modeled using this record:

$$\mathsf{<|\,regs : reg \rightarrow word; memory : addr \rightarrow word; dev\_state :} \ \tau\mathsf{; undefined : bool\,|>}.$$

$\mathsf{ext\_state}$ extends the system state s with a device state ps:

$$\begin{aligned}\mathsf{ext\_state}\ ps\ s \triangleq \mathsf{<|\,regs} &:= s.\mathsf{regs}; \mathsf{memory} := s.\mathsf{memory};\\ \mathsf{dev\_state} &:= (ps, s.\mathsf{dev\_state}); \mathsf{undefined} := s.\mathsf{undefined\,|>},\end{aligned}$$

while $\mathsf{base\_state}$ undoes it:

$$\begin{aligned}\mathsf{base\_state}\ s \triangleq \mathsf{<|\,regs} &:= s.\mathsf{regs}; \mathsf{psrs} := s.psrs; \mathsf{memory} := s.\mathsf{memory};\\ \mathsf{dev\_state} &:= \mathsf{SND}\ s.\mathsf{dev\_state}; \mathsf{undefined} := s.\mathsf{undefined\,|>}.\end{aligned}$$

Here, reg is the type of a register, regs represents the register store, which includes the data registers and special purpose registers such the program counter and processor status registers. I use r_0 to indicate register 0, r_{14} to indicate register 14, etc. pc is used to indicate the program counter. They are all of reg type. memory represents the memory. dev_state represents the state of the devices. The system is in an erroneous state when undefined is set.

Given a state s, is_undefined $s \triangleq s$.undefined. next in Equation 2.1 should implement the execution of the processor core and transit for the device in parallel, as shown in Figure 2.3. They are independent of each other except when the instruction is a command to the device. In this scenario, the processor core commands the device to run mapped_read or mapped_write and reads data from or writes data to the specific device register. It may set undefined based on the results of these operations. At the same time, when running mapped_read or mapped_write the device updates its state. The device finally updates its state again with transit.

2.3.1 Reason about Multiple Devices

To support the modularity when reasoning about systems with multiple devices, I refine the definition in Equation 2.1 and use

$$<|\,\mathsf{next} : \tau\ \mathsf{dev} \rightarrow \mathsf{state} \rightarrow \mathsf{state}; \mathsf{is\_undefined} : \mathsf{state} \rightarrow \mathsf{bool}\,|>$$

to model an embedded system. Similarly, the new step is

$$\mathsf{step}\ p\ n\ s \triangleq \mathsf{if}\ n = 0\ \mathsf{then}\ s\ \mathsf{else}\ \mathsf{next}\ p\ (\mathsf{step}\ p\ (n-1)\ s).$$

The new sys_pred is defined in Figure 2.4.

The theorem in Figure 2.5 establishes that adding a new device does not break a system that was previously working. If a system does not run into an erroneous state in t steps running a program, it will not run into an erroneous state in t steps running the same program with device p_2 plugged in.

It is obvious that the program does not access the addresses mapped to p_2 in these t steps. Otherwise it would have run into an erroneous state. So there is no chance for p_2 to introduce errors to the program in these t steps. In other words, s and ps

are independent of each other in these t steps.

Also, if we can verify a property of a program in a system with only some set of peripherals, the property still holds when more devices are added:

$$\begin{aligned}
&\vdash (\forall a.\ \neg(p_1.\mathsf{mapped}\ a \wedge p_2.\mathsf{mapped}\ a)) \Rightarrow \\
&\quad \mathsf{sys\_pred}\ p_1\ P\ I\ Q \Rightarrow \\
&\quad \mathsf{sys\_pred}\ (\mathsf{comb\_dev}\ p_2\ p_1)\ (P\ \mathsf{o}\ \mathsf{base\_state}) \\
&\qquad (I\ \mathsf{o}\ (\lambda\,(x,y).(\mathsf{base\_state}\ x, \mathsf{base\_state}\ y)))\ (Q\ \mathsf{o}\ \mathsf{base\_state}). \qquad (2.4)
\end{aligned}$$

For any system with device p_1, if sys_predp_1$P\ I\ Q$ holds on it, then it holds for the system with one more device p_2 added, considering only the state components before the plugging in of device p_2.

sys_pred $p_1\ P\ I\ Q$ actually specifies a sequence of transitions of the system. Similar to the theorem in Equation 2.4, in the new system, those components of the system state before the plugging in of device p_2 and state of device p_2 are independent of each other in the sequence. So the sequence specified by sys_pred $p_1\ P\ I\ Q$ is still the same.

```
comb_dev $p_1$ $p_2$ $\triangleq$
<| mapped := $\lambda\, a.\ p_1$.mapped $a \vee p_2$.mapped $a$;
  mapped_read :=
    $\lambda\, a\ s.$ if $p_1$.mapped $a$
          then let $read_1 = p_1$.mapped_read $a$ (FST $s$) in
                (FST $read_1$, (FST o SND) $read_1$, (SND o SND) $read_1$, SND $s$)
          else if $p_2$.mapped $a$
          then let $read_2 = p_2$.mapped_read $a$ (SND $s$) in
                (FST $read_2$, (FST o SND) $read_2$, FST $s$, (SND o SND) $read_2$)
          else (ARB, T, $s$);
  mapped_write :=
    $\lambda\, a\ d\ s.$ if $p_1$.mapped $a$
            then let $write_1 = p_1$.mapped_write $a$ $d$ (FST $s$) in
                  (FST $write_1$, SND $write_1$, SND $s$)
            else if $p_2$.mapped $a$
            then let $write_2 = p_2$.mapped_write $a$ $d$ (SND $s$) in
                  (FST $write_2$, FST $s$, SND $write_2$)
            else (T, $s$);
  transit := $\lambda\, s.$ let $np_1 = p_1$.transit (FST $s$) and $np_2 = p_2$.transit (SND $s$) in
                ($np_1$, $np_2$);
  wellform := $\lambda\, s.\ p_1$.wellform (FST $s$) $\wedge\ p_2$.wellform (SND $s$) |>
```

Figure 2.1: Combination of devices

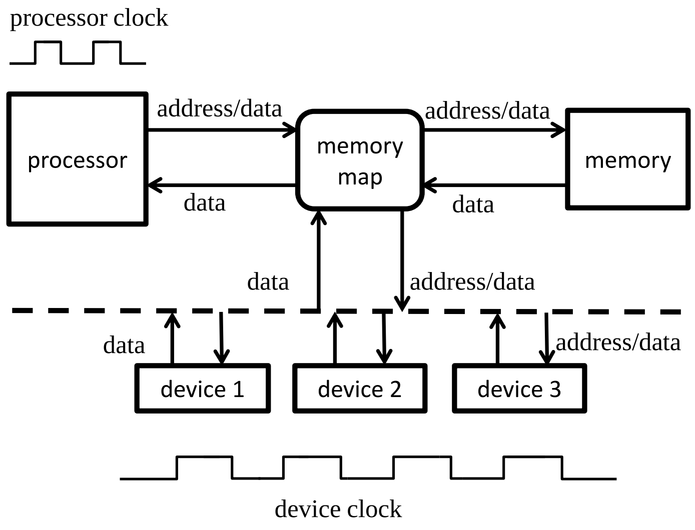

Figure 2.2: Model for system with devices

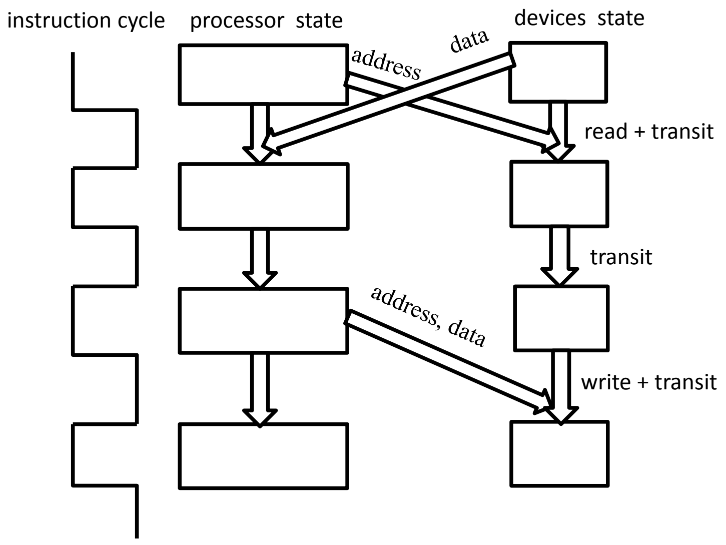

Figure 2.3: Parallel nature of *next*

$$\begin{array}{l}
\mathsf{sys\_pred}\ p\ P\ I\ Q \triangleq \\
\forall\, s.\ P\ s \wedge \neg\mathsf{is\_undefined}\ s \Rightarrow \\
\quad \exists\, t.\ Q\ (\mathsf{step}\ p\ t\ s) \wedge \\
\quad\quad \forall\, n.\ n \le t \Rightarrow I\ (s, \mathsf{step}\ p\ n\ s) \wedge \neg\mathsf{is\_undefined}\ (\mathsf{step}\ p\ n\ s\ )
\end{array}$$

Figure 2.4: sys_pred definition used to reason about embedded systems with multiple devices

$$\begin{array}{l} \vdash (\forall\, a.\ \neg(p_1.\mathsf{mapped}\ \ a \wedge p_2.\mathsf{mapped}\ \ a)) \Rightarrow \\ \quad \forall\, t.\ \neg\mathsf{is\_undefined}\ (\mathsf{step}\ p_1\ \ t\ s) \Rightarrow \\ \quad \neg\mathsf{is\_undefined}\ (\mathsf{step}\ (\mathsf{comb\_dev}\ p_2\ p_1)\ \ t\ (\mathsf{ext\_state}\ ps\ \ s)) \end{array}$$

Figure 2.5: Plugging in a new device does not break the existing system

CHAPTER 3

CORRECTNESS PROOF FOR POLLING-BASED SERIAL PORT DRIVER

In this chapter I demonstrate the approach introduced in Chapter 2 by presenting the correctness proof for a polling-based serial port driver. First, I briefly introduce the communication functions of the serial port. Next, I develop the formal model for the serial port. Next, the assertion formula used to describe the high-level behavior of the serial port is introduced. Then I present the correctness theorems for the polling-based serial port driver, as well as the proof techniques. At the end I summarize the first part of my work, which will be used in the second part.

3.1 Serial Port

A serial port, also known as universal asynchronous receiver/transmitter (UART), is a serial communication device. It is very common in computer systems, especially embedded systems.

The device has two basic communication functions: receiving and transmitting. It uses a clock divider to control the rate of receiving and transmitting, which is called baud rate. For the receiving function, when incoming data is received, it is assembled and placed in a hardware receive buffer register (RBR) if there is room available. Plus, the receive data available (RDA) bit in the line status register (LSR) is set to signal that new data is received. This status bit is reset when data is read from RBR. If the receive buffer is full at the moment of receiving, the buffer overrun error (OE) bit is set in LSR.

For the transmitting function, there is also a hardware transmit holding register (THR), which is used to hold data before it is transmitted out. When this buffer has room available, the transmit holding register empty (THRE) bit in LSR is set to signal the status. This status bit is reset when data is written into THR.

For RDA,OE or THRE, there is a corresponding interrupt, which will become pending when the status bit is set and if it is enabled. This is interrupt-driven operation. If interrupts are disabled, software must query LSR to check the status of the serial port. This is polling-based operation.

3.2 Serial Port Model

I instantiate the abstract device as shown in Section 2.2.1 with a model for the serial port UART0 from an NXP LPC2129 chip [72]. This is a popular embedded processor based on the ARM7TDMI architecture. It targets industrial control applications. I call the model uart0.

My serial port model is conservative: while it does not model all behaviors of the real device, it should be the case that any code that is verified against the model will also work when running on the hardware. Table 3.1 summarizes the model's coverage of the serial port's register set, where $\mathsf{LCR}_{\mathrm{DLAB}}$ stands for divisor latch access bit which controls if registers DLL (divisor latch least significant byte) and DLM (divisor latch most significant byte) are accessible. It is the 7th bit in the LCR (line control register). My model omits modem functionality. Interrupts are not supported in this part of work, and will be supported later. In my model, the internal hardware receiving and transmitting buffers both have fixed size of 1. In LPC2129 the buffer sizes are configurable. It does not model line errors or wire encoding since it is a program model, and it is assumed that whole characters are transmitted. It does not model the break function.

In my model, a register access can lead the system into undefined states in the following scenarios:

1. When the register is not modeled. For example, access of the addresses reserved for the modem function is undefined.
2. When a write-only register is read, or when a read-only register is written.
3. When a reserved bit is written.
4. When data corruption may occur. For example the receiving buffer register is read when its value is indeterminate.

The state of such a serial port model is represented with a record:

$$\langle|\,\mathsf{RBR} : \mathsf{bool[8]};\ \mathsf{THR} : \mathsf{bool[8]};\ \mathsf{SCR} : \mathsf{bool[8]};\ \mathsf{DLL} : \mathsf{bool[8]};\ \mathsf{DLM} : \mathsf{bool[8]};$$
$$\mathsf{LCR}_{\mathrm{DLAB}} : \mathsf{bool};\ \mathsf{LSR}_{\mathrm{RDR}} : \mathsf{bool};\ \mathsf{LSR}_{\mathrm{OE}} : \mathsf{bool};\ \mathsf{LSR}_{\mathrm{THRE}} : \mathsf{bool};\ \mathsf{LSR}_{\mathrm{TEMT}} : \mathsf{bool};$$
$$\mathsf{clock} : \mathsf{num};\ \mathsf{in} : \mathsf{num} \to \mathsf{bool[8]\ option};\ \mathsf{out} : \mathsf{num} \to \mathsf{bool[8]\ option}\,|\rangle.$$

Here, byte is the type for 8-bit byte. num is the type for the natural number. Note that registers LCR and LSR (line status register) are broken down into Boolean flags. Access of FCR (FIFO control register) is modeled as side effect only. THR and out form the output queue, and RBR and in form the input queue.

One important feature is that the speed of the serial port is parameterized relative to the core speed. I am not modeling the exact baud rate, but in a similar fashion I use the 16-bit word value from DLM and DLL as a clock divisor b unless its value is zero, in which case b is set to be 1. For a serial port state ps, get_divisor ps returns b. The serial port only performs meaningful state transition every b cycles. To do so, clock is incremented for each instruction cycle. It will be reset to zero when it reaches b, at the moment the device transmits and receives data, updates its registers, and shifts its input and output streams. At other moments when $0 < \mathsf{clock} < b - 1$, the serial port only updates clock for book-keeping purposes. However, memory-mapped accesses from the processor core can occur at any clock value. The effect of these accesses are visible when a new cycle begins.

The incoming and outgoing data streams are modeled as functions in and out from natural numbers to byte option. An option type has two constructors, THE and NONE. THE x wraps x into the particular option type, while NONE indicates nothing is wrapped, which is suitable to describe that at some moments the input or output stream are idle with no characters transmitted. With every b cycles of instruction execution, the two streams will shift. The new value for in is

$$\lambda\, t.\ \mathsf{in}\ (t+1),$$

and the new value for out is

$$\lambda\, t.\ \mathsf{if}\ t = 0\ \mathsf{then}\ d\ \mathsf{else\ out}\ (t-1)$$

where d is the character just sent out.

3.3 High-Level View of the Serial Port Behavior

I describe the high-level behavior of the serial port device in terms of strings extracted from the output queue and input queue. Only nonempty strings are considered. The predicates are defined in Figure 3.1. Suppose hd, tl return the first character and the tail of a string, respectively, and T stands for Boolean value true. Serial port states which are not well-defined are excluded by the uart_wellform function. An input stream can be shifted by cutStream. For the transmit function, outString *s os* describes that *s* is the most recent string in the output stream *os*. sentString *s ps* describes that *s* is the most recent string sent out by the processor in the serial port state *ps*. For the receive function, inString *s is* describes that *s* is the string in the input stream *is*. inputString *s ps* describes that *s* is the next string to be received by the processor in the serial port state *ps* if no buffer overrun ever occurs. shifted ps_2 ps_1 is a weak safety invariant for the receive function. It states that the RBR and income stream in state ps_2 resulted from ps_1 after some cycles of operation. inputStream *str ps* describes that *str* is the extended income stream of *ps* in which RBR is appended at the head of the income stream if there is data in RBR.

For a serial port state *ps*, sentString s_{rx} *ps* associates its receive state with string s_{rx}, and sentString s_{tx} *ps* associate its transmit state with string s_{rx}. Using strings, it is intuitive to assert the behavior of the serial port driver. For example, s_{rx} does not change if there is no buffer overrun over an autonomous transition of the serial port.

3.4 Correctness of a Serial Port Driver

I started with a freely available driver for the LPC2129's UART0 that is implemented in C, and compiled it to ARM assembly using GCC 4.1.1. I made one change to the compiler's output, which was to change the "bx" instruction that implements a return-from-function to a "mov" instruction. I did this because "bx" is not modeled in the ARM tool that does not support the THUMB mode. I proved full correctness for three functions which interact with device registers: the putch function which transmits a character by writing it to THR, the getch function which attempts to read a character from RBR, and the getchwW function which performs a blocking read from RBR. The code is shown in Figure 3.2.

3.4.1 Serial Port Model Soundness

The correctness claim is based on the correctness claims of other components. I assume that the ARM model in HOL4 is correct (this ARM model has been used in several projects, and an earlier version was verified against a specific instance of the ARM hardware). Then I depend on the fact that the abstract device model attached to the ARM model is sound as shown in Section 2.3. The soundness of the serial port model is proved in the process. For example, I have proved the following properties regarding the transmitting and receiving functions, among others:

1. No character will be appended to the output under any of the following conditions:
   (a) no memory-mapped read or write occurs,
   (b) a read occurs,
   (c) a write occurs but the THR register is not accessible or not written, and the FCR register is not written (to reset the transmission queue).

   In fact, the only way to append a character to the output is to write to the THR register when thre set. The theorem is shown in Figure 3.3.
2. The input string is not changed under any following conditions:
   (a) between two ticks of the serial port clock,
   (b) the serial port clock ticks, but there is no incoming character on the wire or the RBR is empty,
   (c) a memory-mapped write occurs, and the FCR register is not written (to reset the transmission queue),
   (d) a memory-mapped read occurs, and RBR is not read.

   The only way to read the head of the string is to read the RBR when it is not empty. The theorem is shown in Figure 3.4.

3.4.2 Memory Safety and Control Flow Integrity

To prove the full correctness we need to prove memory safety and control flow integrity of the driver code. These properties are a useful part of the safety specification, and also are important for proof management. Memory safety requires that only a given range of registers in the ARM core and memory is accessed. This implies compliance to the calling convention. So it is useful when proving the callers of the driver functions. It also implies the separation of instruction memory, which is essential to prove control flow integrity.

Ideally, the addresses or registers accessed in an ARM instruction could be known when it is decoded. Here, I use a different approach by examining the change of content in the memory and registers:

$$\begin{aligned}&\mathsf{sep\_mem}\ \mathit{addrMap}\ s_1\ s_2 \triangleq\\ &\quad\forall x.\ \mathit{addrMap}\ x \Rightarrow (\ s_2.\mathsf{memory}\ (\mathsf{addr30}\ x) =\ s_1.\mathsf{memory}\ (\mathsf{addr30}\ x)),\\ &\mathsf{sep\_reg}\ \mathit{regMap}\ s_1\ s_2 \triangleq \forall x.\ \mathit{regMap}\ x \Rightarrow (s_2.\mathsf{regs}\ x = s_1.\mathsf{regs}\ x).\end{aligned}$$

sep_mem accSet s1 s2 checks two system states s1 and s2 to see if any address not in the set addrSet have the same content. sep_reg regSet s1 s2 does the same thing for the registers across two states. Since the access which could cause side effects is limited to access of memory-mapped device registers, which is already taken care of, this approach serves our purpose well. However, these two predicates are rather naive, an embedding of separation logic here would have be cleaner.

Control flow integrity specifies that only certain sequences of PC values can occur in the execution. For example, when putch is busy waiting, it strictly follows the loop. Control flow integrity is necessary to prove the loop invariant, or generally any data flow, and thus helps us to sequentially compose the theorems about segments of the execution together to prove the final theorem. In the final theorem I did not include the stepwise specification of the control flow.

3.4.3 Correctness of the Serial Port Driver

I proved the full correctness theorems for three functions: putch, getch, and getchw. putch first waits for thre being set. It will then copy the byte from register r0 to THR. getch copies the byte from RBR to register r0 if $\mathsf{LSR}_{\mathrm{RDR}}$ is set. Otherwise it returns 0xff. getchw first waits for $\mathsf{LSR}_{\mathrm{RDR}}$ being set. It will then copy the byte from RBR to register r0. Note that if getchw is used to receive a string, characters may be dropped if the serial port is too fast.

The correctness property includes both liveness and safety properties. For all three functions, the basic liveness property states that the function will return to its caller. In addition, the basic safety property states that memory safety is observed, the operating configuration of the serial port device is not changed in terms of its speed (described by the slow-down factor) and the controlling bit $\mathsf{LCR}_{\mathrm{DLAB}}$, and the system do not run into any erroneous state. Functional correctness is proved for putch, getch and getchw:

1. putch successfully appends the character from $r0$ to the string already sent out in the output queue. The basic safety and liveness properties hold in the process. The theorem is shown in Figure 3.5.
2. getch successfully reads a character from the input queue or return 0xff. The basic safety and liveness properties hold in the process. The theorem is shown in Figure 3.6.
3. If there is a string in the input queue, and the serial port is slow enough, the function getchw will successfully read the next character from the input stream. In the process, no overrun error occurs to the serial port, and the basic safety and liveness properties hold. The theorem is shown in Figure 3.7.

The correctness of getchw depends on the speed of the serial port device relative to the ARM core, and the latency caused by the driver code. The driver code must be efficient enough and the serial port must be slow enough so that no buffer overrun error can occur. My approach allows this constraint to be expressed, while the previous work does not [2]. The authors discussed receiver buffer overrun issue from a programmer's point of view and suggested three approaches. One of them is to do a worst case execution time (WCET) analysis of interrupt handler and the device driver and derive the latency in the processing of received data. The latency can be used to derive the maximum baud rate of the serial port.

However, the authors modeled the device like a thread concurrent to the processor, and did not introduce a clock to their serial port model. As a result, timing related properties such as buffer overrun cannot even be expressed in their model.

The tight timing properties in the theorem in Figure 3.7 will be helpful when proving the string-level receiving function, which calls getchw repetitively. The string can be retrieved completely without overrun, as long as the interval between the consecutive return and entry of getchw is bounded by *delay*, which is bounded by the difference between the clock divisor of the serial port and the latency introduced in getchw, which is 9 instruction cycles. This guarantees that oe is not set when getchw is entered the next time, thus the precondition of getchw is met.

3.4.4 Proof Method

All the theorems about the execution I proved are in the form of sys_pred uart0$P\ I\ Q$. I use LSB to extract the least significant byte from a register, which is 32 bits wide in the ARM model that I use. c::str appends a character c to the head of a string str. The modifiable register sets are defined in putchReg = getchwReg = {r0, r1, r3, pc} for putch and getchw, and getchReg = {r0, pc} for putch, respectively. Set lpcMapped indicates all the memory addresses which are mapped to devices in a LPC2129 system-on-chip (SoC).

putch and getchw work in the polling mode by testing for certain conditions with a busy-waiting loop. Termination of the loop depends on the state of the device, and needs to be proved. One difficulty in proving loop termination is that the device speed is parameterized.

The termination theorem is a sys_pred predicate, which is defined in Equation (2.2). In the proof, I provide the witness for the existentially qualified t, then use induction on time n.

Use putch as an example: I break the execution sequence into three parts. With each part I prove a correctness lemma in the form of sys_pred uart0$P\ I\ Q$. The control flow and data flow assertions are encoded in I. The first part is the busy waiting until thre is set in the serial port. The length of this waiting depends on the speed of the serial port and the serial port state at the point of entry of putch. When thre is set, the program counter could be at any instruction of the loop. The second part is the break of the loop from the point where thre is set to the exit of the loop. The third part is the sequential execution to copy the character to the register THR and return. In classic cases when no devices are concerned, the first two parts are treated as one, since it is at a static instruction point that the condition triggering the break of the loop is met.

putch, getch and getchw are used to implement string-level transmitting and receiving functions. Proving the correctness of these functions does not need to work at the level of device details. Sophisticated program logic [29, 60, 61, 93] may be needed to deal with scaling and more complicated control flow. The state of the device can be trivially plugged into the proof based on the theorem in Equation 2.4. The theorems I proved already imply the calling convention. It should not be difficult to translate them into appropriate format and integrate them into high-level proof.

3.5 Summary

This concludes the first part of my work: the correctness proof of polling-based serial port drivers, putch, getch and getchW. Even though the drivers proved here are small in terms of code size, the proofs show the strength of my approach.

The abstract device model provides an interface to introduce realistic device models to an ISA model. With it I obtained some general theorems about multiple devices. These theorems are rather intuitive, and can be thought of as a sanity check to the abstract device model.

The abstract device model facilitates automation. The automation scripts developed using it can be applied to any concrete device models as long as they instantiate this abstract model. The reason is that the abstract device model is a type inside HOL, and a concrete device model is an instance of this type. What is more, combining multiple concrete device models results in a single device, which is still only one instance of the abstract device model.

The clock divider allows timing properties to be expressed and reasoned about. The insight is that devices are always slower than the processor. So the speed of a device is modeled relative to the speed of the processor, and time is measured in terms of processor instruction cycles. In this approach devices run in parallel to the processor core. There is no interleaving between them.

Table 3.1: Serial port model coverage. R indicates read-only registers; W indicates write-only. RW indicates no restriction on access. The first four columns are from data in the LPC2129 manual.

Register	Address offset	Function	Access	When is read undefined?	When is write undefined?	Side-effect of read	Side-effect of write
RBR	0	Receiver buffer when $\neg\mathsf{LCR}_{\mathrm{DLAB}}$	R	No data received	Never	Reset $\mathsf{LSR}_{\mathrm{RDR}}$	None
THR	0	Transmit holding when $\neg\mathsf{LCR}_{\mathrm{DLAB}}$	W	Never	No room for transmission	None	Reset $\mathsf{LSR}_{\mathrm{THRE}}$
DLL	0	Divisor latch LSB when $\mathsf{LCR}_{\mathrm{DLAB}}$	RW	Never	Never	None	None
DLM	4	Divisor latch MSB when $\mathsf{LCR}_{\mathrm{DLAB}}$	RW	Never	Never	None	None
FCR	8	FIFO control	W	Always	Overwrite reserved bits or disable FIFOs	None	Reset transmission or receiving queue and flags
LCR	12	Line control	RW	Never	Never	None	Assign $\mathsf{LCR}_{\mathrm{DLAB}}$ flag
LSR	20	Line status	R	Never	Always	Reset $\mathsf{LSR}_{\mathrm{OE}}$	None
SCR	28	scratch pad	RW	Never	Never	None	None

$$
\begin{aligned}
&\mathsf{uart\_wellform}\ ps \triangleq (\neg ps.\mathsf{LSR}_{\mathrm{TEMT}} \lor ps.\mathsf{LSR}_{\mathrm{THRE}}) \land\\
&\qquad (\neg(ps.\mathsf{clock} = 0) \lor ps.\mathsf{LSR}_{\mathrm{THRE}}) \land\\
&\qquad ps.\mathsf{clock} < \mathsf{get\_divisor}\ ps\\
&\mathsf{cutStream}\ n\ s \triangleq \lambda\, x.\ s\ (n + x)\\
&\mathsf{outString}\ s\ os \triangleq \exists\, n.\ (os\ n = \mathsf{SOME}\ (\mathsf{hd}\ s)) \land (\forall\, l.\ l < n \Rightarrow (os\ l = \mathsf{NONE})) \land\\
&\qquad \mathsf{outString}\ (\mathsf{tl}\ s)\ (\mathsf{cutStream}\ (n+1)\ os)\\
&\mathsf{sentString}\ s\ ps \triangleq \mathsf{if}\ \neg ps.\mathsf{thre}\\
&\qquad \mathsf{then}\ (ps.\mathsf{THR}\ = h) \land \mathsf{outString}\ (\mathsf{tl}\ s)\ ps.\mathsf{out}\\
&\qquad \mathsf{else}\ \mathsf{outString}\ s\ ps.\mathsf{out}\\
&\mathsf{inString}\ s\ is\ \triangleq \exists\, n.\ (is\ n = \mathsf{SOME}\ h) \land (\forall\, l.\ l < n \Rightarrow (is\ l = \mathsf{NONE})) \land\\
&\qquad \mathsf{inString}\ (\mathsf{tl}\ s)\ (\mathsf{cutStream}\ (n+1)\ is)\\
&\mathsf{inputString}\ s\ ps\ \triangleq \mathsf{if}\ ps.\mathsf{rdr}\\
&\qquad \mathsf{then}\ (ps.\mathsf{RBR}\ = \mathsf{hd}\ s) \land \mathsf{inString}\ (\mathsf{tl}\ s)\ ps.\mathsf{in}\\
&\qquad \mathsf{else}\ \mathsf{inString}\ s\ ps.\mathsf{in}\\
&\mathsf{shifted}\ ps_1\ ps_2 \triangleq \exists\, n.\ (ps_2.\mathsf{in} = \mathsf{cutStream}\ n\ ps_1.\mathsf{in}) \land\\
&\qquad (\mathsf{if}\ ps_2.\mathsf{rdr}\\
&\qquad\ \mathsf{then}\ ps_1.\mathsf{LSR}_{\mathrm{RDR}} \land (ps_2.\mathsf{RBR} = ps_1.\mathsf{RBR}) \lor\\
&\qquad\qquad \exists\, k.\ k < n \land (ps_1.\mathsf{in}\ k = \mathsf{SOME}\ ps_2.\mathsf{RBR})\\
&\qquad\ \mathsf{else}\ \mathsf{T})\\
&\mathsf{inputStream}\ str\ ps \triangleq\\
&\quad \mathsf{if}\ ps.\mathsf{rdr}\\
&\quad \mathsf{then}\ str = \lambda\, x.(\mathsf{if}\ x = 0\ \mathsf{then}\ \mathsf{SOME}\ ps.\mathsf{RBR}\ \mathsf{else}\ ps.\mathsf{in}\ (x-1))\\
&\quad \mathsf{else}\ str = \lambda\, x.(\mathsf{if}\ x = 0\ \mathsf{then}\ \mathsf{NONE}\ \mathsf{else}\ ps.\mathsf{in}(x-1))
\end{aligned}
$$

Figure 3.1: Formula used to assert the serial port state

```
<Putch>:                <Getch>:                  <GetchW>:
  ldr  r2, #0xe000c000    ldr    r2, #0xe000c000    ldr  r2, #0xe000c000
  ldrb r3, [r2, #20]      ldrb   r3, [r2, #20]      ldrb r3, [r2, #20]
  tst  r3, #32            tst    r3, #1             tst  r3, #1
  beq  <Putch>            ldrneb r3, [r2]           beq  <GetchW>
  and  r0, r0, #255       mvn    r0, #0             ldrb r0, [r2]
  strb r0, [r2]           andne  r0, r3, #255       mov  pc, lr
  mov  pc, lr             mov    pc, lr
```

Figure 3.2: ARM assembly code for putch, getch and getchw

$$
\begin{aligned}
&\vdash (\forall\, s\ ps.\ \mathsf{sentString}\ s\ ps \Rightarrow \mathsf{sentString}\ s\ (\mathsf{uart0.transit}\ ps)) \wedge \\
&\quad (\forall\, s\ ps\ addr.\ \mathsf{sentString}\ s\ ps \Rightarrow \\
&\qquad\qquad\quad \mathsf{sentString}\ s\ (\mathsf{SND}\ (\mathsf{SND}\ (\mathsf{uart0.mapped\_read}\ addr\ ps)))) \wedge \\
&\quad (\forall\, s\ d\ addr\ ps.\ (ps.\mathsf{LCR}_{\mathrm{DLAB}} \vee \neg(addr = 0xE000C000)) \wedge \\
&\qquad\qquad\qquad \neg(addr\ = 0xE000C008) \wedge \mathsf{sentString}\ s\ ps \Rightarrow \\
&\qquad\qquad\qquad \mathsf{sentString}\ s\ (\mathsf{SND}\ (\mathsf{uart0.mapped\_write}\ addr\ d\ ps))) \wedge \\
&\quad (\forall\, s\ c\ ps.\ \neg ps.\mathsf{LCR}_{\mathrm{DLAB}} \wedge ps.\mathsf{thre}\ \wedge (addr = 0xE000C000) \wedge \mathsf{sentString}\ s\ ps \Rightarrow \\
&\qquad\quad \mathsf{sentString}\ (c :: s)\ (\mathsf{SND}\ (\mathsf{uart0.mapped\_write}\ addr\ (\mathsf{Byte}\ c)\ ps)))
\end{aligned}
$$

Figure 3.3: Safety property for the transmitting function in the serial port model

$$
\begin{aligned}
\vdash\ & (\forall\, s\ ps.\ (ps.\mathsf{clock} + 1 < \mathsf{get\_divisor}\ s \lor (ps.\mathsf{in}\ 0 = \mathsf{NONE}) \lor \lneg ps.\mathsf{LSR}_{\mathrm{RDR}}) \land \\
&\qquad\quad \mathsf{inputString}\ s\ ps \Rightarrow \\
&\qquad\quad \mathsf{inputString}\ s\ (\mathsf{uart0.transit}\ ps)) \land \\
& (\forall\, ps\ addr.\ \mathsf{inputString}\ s\ ps \land (\lneg(addr = 0xE000C000) \lor \lneg ps.\mathsf{LSR}_{\mathrm{RDR}} \Rightarrow \\
&\qquad\qquad \mathsf{inputString}\ s\ (\mathsf{SND}\ (\mathsf{SND}\ (\mathsf{uart0.mapped\_read}\ addr\ ps)))) \land \\
& (\forall\, s\ d\ addr\ ps.\ \lneg(addr = 0xE000C008) \land \mathsf{inputString}\ s\ ps \Rightarrow \\
&\qquad\qquad\quad \mathsf{inputString}\ s\ (\mathsf{SND}\ (\mathsf{uart0.mapped\_write}\ addr\ d\ ps))) \land \\
& (\forall\, ps\ h\ t.\ \lneg ps.\mathsf{LCR}_{\mathrm{DLAB}} \land \mathsf{inputString}\ (h :: t)\ ps \land ps.\mathsf{LSR}_{\mathrm{RDR}} \Rightarrow \\
&\qquad\quad \mathsf{inputString}\ t\ (\mathsf{SND}\ (\mathsf{SND}\ (\mathsf{uart0.mapped\_read}\ 0xE000C000\ ps))) \land \\
&\qquad\quad \lneg(\mathsf{SND}\ (\mathsf{SND}\ (\mathsf{uart0.mapped\_read}\ 0xE000C000\ ps))).\mathsf{LSR}_{\mathrm{RDR}})
\end{aligned}
$$

Figure 3.4: Safety property for the receiving function in the serial port model

$$\vdash \mathsf{sys\_pred}\ \mathsf{uart0}\ (\mathsf{P\_putch}\ reAddr\ c\ str)\ \mathsf{I\_putch}\ (\mathsf{Q\_putch}\ reAddr\ c\ str)$$

where the precondition, invariant and postcondition are

$$\begin{aligned}
&\mathsf{P\_putch}\ reAddr\ c\ str\ s \triangleq \\
&\quad \mathsf{code\ putch}\ s \wedge (s.\mathsf{regs\ r14} = reAddr) \wedge (s.\mathsf{regs\ pc} = 0x28c) \wedge \\
&\quad \mathsf{uart0.wellform}\ (s.\mathsf{dev\_state}) \wedge \mathsf{sentString}\ str\ (s.\mathsf{dev\_state}) \wedge \\
&\quad (\mathsf{LSB}\ (s.\mathsf{regs\ r0}) = c) \wedge \neg s.\mathsf{dev\_state.LCR}_{\mathrm{DLAB}}, \\
&\mathsf{I\_putch}\ (s, ns) \triangleq \\
&\quad \mathsf{sep\_mem\ safePutchAddr}\ s\ ns \wedge \mathsf{sep\_reg\ safePutchReg}\ s\ ns \wedge \\
&\quad (\mathsf{get\_divisor}\ ns.\mathsf{dev\_state} = \mathsf{get\_divisor}\ s.\mathsf{dev\_state}) \wedge \\
&\quad \mathsf{uart0.wellform}\ ns.\mathsf{dev\_state} \wedge \neg ns.\mathsf{dev\_state.LCR}_{\mathrm{DLAB}}, \\
&\mathsf{Q\_putch}\ reAddr\ c\ str\ ms \triangleq \\
&\quad \mathsf{sentString}\ (c :: str) ms.\mathsf{dev\_state} \wedge (ms.\mathsf{regs\ pc} = reAddr)
\end{aligned}$$

Figure 3.5: Correctness theorem for putch

$$\vdash \mathsf{sys\_pred\ uart0}\ (\mathsf{P\_getch}\ \mathit{reAddr}\ \mathit{strm})\ \mathsf{I\_getch}\ (\mathsf{Q\_getch}\ \mathit{reAddr})$$

where the precondition, invariant and postcondition are

$$\begin{aligned}
&\mathsf{P\_getch}\ \mathit{reAddr}\ \mathit{strm}\ s \triangleq\\
&\quad \mathsf{code\ putch}\ s \wedge (s.\mathsf{regs\ pc} = 0x314) \wedge (s.\mathsf{regs\ r14} = \mathit{reAddr}) \wedge\\
&\quad \mathsf{uart0.wellform}\ (s.\mathsf{dev\_state}) \wedge \mathsf{inputStream}\ \mathit{strm}\ s.\mathsf{dev\_state} \wedge\\
&\quad \neg(s.\mathsf{dev\_state}).\mathsf{LCR}_{\mathrm{DLAB}},\\
&\mathsf{I\_getch}\ (s, ns) \triangleq\\
&\quad \mathsf{sep\_mem\ safeGetchAddr}\ s\ ns \wedge \mathsf{sep\_reg\ safeGetchReg}\ s\ ns \wedge\\
&\quad (\mathsf{get\_divisor}\ (ns.\mathsf{dev\_state}) = \mathsf{get\_divisor}\ (s.\mathsf{dev\_state})) \wedge\\
&\quad \mathsf{uart0.wellform}\ (ns.\mathsf{dev\_state}) \wedge \neg(ns.\mathsf{dev\_state}).\mathsf{LCR}_{\mathrm{DLAB}} \wedge\\
&\quad \mathsf{shifted}\ ns.\mathsf{dev\_state} s.\mathsf{dev\_state},\\
&\mathsf{Q\_getch}\ \mathit{reAddr}\ \mathit{strm}\ ms \triangleq\\
&\quad ((ms.\mathsf{regs\ r0} = 0xff) \vee \exists k.\ \mathit{strm}\ k = \mathsf{SOME}\ (\mathsf{LSB}\ (ms.\mathsf{regs\ r0}))) \wedge\\
&\quad (ms.\mathsf{regs\ pc} = \mathit{reAddr})
\end{aligned}$$

Figure 3.6: Correctness theorem for getch

$$\vdash \neg(str\ = []) \wedge 9 + delay < div \Rightarrow$$
$$\quad \mathsf{sys\_pred\ uart0}\ (\mathsf{P\_getchw}\ str\ div\ reAddr)\ (\mathsf{I\_getchw}\ div)$$
$$\quad\quad (\mathsf{Q\_getchw}\ delay\ str\ div\ reAddr)$$

where the precondition, invariant and postcondition are

$\mathsf{P\_getchw}\ str\ div\ reAddr\ s \triangleq$
$\quad \mathsf{code\ getchw}\ s \wedge (s.\mathsf{regs\ r14} = reAddr) \wedge (s.\mathsf{regs\ pc} = 0x334) \wedge$
$\quad \mathsf{uart0.wellform}\ (s.\mathsf{dev\_state}) \wedge (div = \mathsf{get\_divisor}\ (s.\mathsf{dev\_state})) \wedge$
$\quad \mathsf{inputString}\ str\ (s.\mathsf{dev\_state}) \wedge \neg(s.\mathsf{dev\_state}).rdr \wedge$
$\quad \neg(s.\mathsf{dev\_state}).\mathsf{oe} \wedge \neg(s.\mathsf{dev\_state}).\mathsf{LCR}_{\mathrm{DLAB}},$
$\mathsf{I\_getchw}\ div\ (s, ns) \triangleq$
$\quad \mathsf{sep\_mem\ safeGetchWAddr}\ s\ ns \wedge \mathsf{sep\_reg\ safeGetchWReg}\ s\ ns \wedge$
$\quad (\mathsf{get\_divisor}\ (ns.\mathsf{dev\_state}) = \mathsf{get\_divisor}\ (s.\mathsf{dev\_state})) \wedge$
$\quad \mathsf{uart0.wellform}\ (ns.\mathsf{dev\_state}) \wedge \neg(ns.\mathsf{dev\_state}).\mathsf{oe} \wedge \neg(ns.\mathsf{dev\_state}).\mathsf{LCR}_{\mathrm{DLAB}},$
$\mathsf{Q\_getchw}\ delay\ str\ div\ reAddr\ ms \triangleq$
$\quad (\mathsf{LSB}\ (\ ms.\mathsf{regs\ r0}) = \mathsf{HD}\ str) \wedge \mathsf{inputString}\ (\mathsf{TL}\ str)(ms.\mathsf{dev\_state}) \wedge$
$\quad \neg(ms.\mathsf{dev\_state}).\mathsf{LSR}_{\mathrm{RDR}} \wedge (ms.\mathsf{dev\_state}).\mathsf{clock} + delay + 1 < div \wedge$
$\quad (ms.\mathsf{regs\ pc} = reAddr)$

Figure 3.7: Correctness theorem for getchW

CHAPTER 4

ARM SOC MODEL

In this chapter I first explain the rationale behind my approach, then give a brief overview of interrupt handling in ARM. Next, a monadic ARM ISA model and the accompanying Hoare logic are introduced. Both of these pieces of research were done at the University of Cambridge and implemented in HOL. Together I will call them the Cambridge ARM model. At the end of this chapter I explain my extension to the ARM model and the Hoare logic. The extension is necessary for proving correctness of interrupt-driven device drivers.

The Cambridge ARM model and my extension can be viewed in three layers:

1. The bottom layer contains the logic model of the ARM core. Here, registers, coprocessors and memory are defined as the elements in the system state. Primitive operations such as memory access are defined on these elements.
2. The middle layer contains the ARM instruction semantics. At first the instruction execution is defined over the system state. Then the semantics are presented as theorems. The theorems are automatically derived.
3. The top layer contains the Hoare logic. Here, the system state is represented as a set. Some separation logic style operations such as separation conjunction are defined using this set. The operational semantics theorems are lifted into Hoare triples. The precondition and postcondition of these Hoare triples are in the form of separation conjunctions. These Hoare triples are derived from the theorems at the middle layer automatically.

The sections on the Cambridge ARM model and my extensions are structured similarly so the difference can be easily seen. I use a running example in this chapter to explain both the Cambridge ARM model and my extension: the semantics of `STRB r2, [r3]` in `svc` mode. This instruction stores the byte in register `r2` at the memory location pointed to by the pointer in register `r3`. It is explained in

Section 1.3.4.

4.1 Overview

Interrupts cause concurrency issues. This concurrency is asymmetric in the sense that the main program can be interrupted by interrupt service routines (ISRs), but not vice versa. This means that we can verify ISRs without considering the effect of the main program. To verify the main program, the effect of ISRs has to be considered.

From the perspective of the main program, if an interrupt is not disabled, its ISR can run at any program point. It may change some register values and contents at some memory locations, and consumes time. The last one is especially important. Interrupts are designed to give fast response, and low latency of interrupt handling is often assumed by programs. For example, for the serial port in interrupt-driven mode, if we do not want to lose some incoming data, the latency in handling the serial port interrupts must be bounded.

Assume there is only one ISR used for handling interrupts requests. In a simple scenario, the ISR does not access the resources such as registers or memory locations of the main program. If we do not consider time here, the effect of the seemingly random firing of interrupts to the execution of an instruction of the main program can be described using invariants involving the registers and memory locations accessed by the ISR. When no interrupt is pending, any invariant is certainly preserved per the frame rule. When some interrupt becomes pending, the contract of the ISR constrains what its invariant can be. So interrupt handling weakens the properties which can be proved about the main program. Examining only the scenario when interrupts are pending is all we need to specify the invariant.

Taking into account the effect of time complicates the analysis of the invariant. An invariant involving time is not necessarily preserved even without interrupt handling. We have to look at both cases, with or without interrupts pending to specify the invariant.

If there are read/write or write/write conflicts among the resources accessed by the ISR and the main program, the contracts of the ISR and the main program instruction constrain what such an invariant can be.

I use a bottom-up approach in verifying an interrupt-driven device driver. The building block is the Hoare triple describing the semantics for each instruction, with the effects of ISR included. Then, I can use some known techniques [60] in the program logic to prove properties about a program in terms of Hoare triples, even though most of these techniques are designed to work with sequential programs.

4.2 ARM Interrupt Handling

When the interrupt mask is off in CPSR, as $\mathsf{aS1}\ \mathsf{PSR}_{\mathrm{I}}\ \mathsf{F}$ in the assertion, the processor can handle interrupt requests. The hardware will perform the following standard tasks with an interrupt request:

1. The processor will complete the execution of the instruction which is in the executing stage when the interrupt request occurs.
2. The processor switches to the IRQ (interrupt) mode. This is reflected in the mode field of CPSR. It is aMD 18 in the assertion.
3. RegisterCPSR of the previous processor mode is saved to the SPSR in the IRQ mode.
4. PC is saved to LR in the IRQ mode. Because of the pipeline, this address is the address of the previously executing instruction plus 8.
5. Interrupts are disabled. This is reflected in the IRQ bit of the CPSR in the IRQ mode. It is $\mathsf{aS1}\ \mathsf{PSR}_{\mathrm{I}}\ \mathsf{T}$ in the assertion.
6. PC is set to a specific address to start the interrupt handling. This address is the interrupt entry in the vector table. In my machine model, the vector table is not used, and this address points to the first instruction of the interrupt handler. Since only one device, serial port exists in the system, the interrupt handler is the ISR.

Now ISR will handle the interrupt requests. In addition to doing work specific to the interrupt requests, ISR needs to perform a context switch. There are common idioms for doing this. The idioms are different depending on which interrupt handling scheme is being used.

In this work I focused on the simple nonnested interrupt handling scheme, in which interrupt is disabled until the ISR is finished. In this scheme, at its entry the ISR needs to save context:

1. The nonbanked registers of the IRQ mode include R_1 through R_{12}. The subset of them to be used need to be saved.
2. LR or $\mathsf{LR} - 4$ needs to be saved. When the ISR is finished, PC needs to be set to LR - 4 so that the previously interrupted task can continue.

After saving the context tasks specific to the interrupt request are performed. Then the ISR returns, and restores the context including nonbanked registers and CPSR. PC is set to the saved PC minus four.

4.3 Cambridge ARM Model

The Cambridge ARM model [1, 36, 37, 60] used in this part of research has much more detail than the one [32, 33] used in the proof of the polling-based driver in Chapter 3. For example, it is parameterized with different versions of the ARM ISA, and models the exceptions of the ARM instruction cycle. It also comes with Hoare logic to support reasoning about ARM assembly programs. The instruction semantics theorems can be automatically derived at both the operational semantics level and the Hoare logic level.

4.3.1 ARM Core Model

In the Cambridge ARM model, there is no device or memory-mapped device access, as shown in the memory read/write definitions. The memory read primitive is defined below:

$$\begin{aligned}
&\mathsf{read\_mem}\ ii\ (memaddrdesc, size) \triangleq \mathsf{if}\ size \notin \{1;2;4;8\} \\
&\quad \mathsf{then\ errorT}\ \text{``read\_mem: size is not 1, 2, 4 or 8''} \\
&\quad \mathsf{else}\ (\mathsf{let}\ address = memaddrdesc.\mathsf{paddress\ in} \\
&\qquad \mathsf{forT}\ 0\ (size - 1)\ (\lambda\, i.\ \mathsf{read\_mem1}\ ii\ (address + n2w\ i))). \qquad (4.1)
\end{aligned}$$

Here, the memory read command checks the exception,[1] and reads the bytes from the memory. Note that read_mem,read_mem1, errorT, and forT are all monads.

[1]It is introduced by the way in which the model is constructed, not from the ARM ISA specification. A valid ARM memory access instruction will not produce this exception

read_mem1 reads a single byte from the memory, errorT throws an exception, and forT implements a loop iteration.

4.3.2 ARM Instruction Semantics

At the middle layer the next-state transition function ARM_NEXT is defined. The semantics for an ARM instruction are presented as theorems, which describe the semantics of valid execution of an instruction by stating what values are read from which registers or memory locations, and which registers or memory locations are updated with what values as well as the conditions which preclude exceptions. For a conditional instruction, two theorems are provided to cover two branches.

ARM instruction STRB r2, [r3] stores the least significant byte from register 2 to the address stored in register 3. This instruction can execute in more than one mode. The formal semantics of its execution in the svc mode in the Cambridge ARM model are shown in Figure 4.1. Here, the conditions and the new system state are expressed using the access functions. Functions with name of pattern ARM_READ_x s read element x from the system state s. Similarly, functions with name of pattern ARM_WRITE_x s update element x in the system state s and return the result. For example, ARM_READ_REG 15 s returns the word from register 15 (PC). The version of the ARM ISA is specified as $v4t$. ARM_NEXT is the next-state transition function of the ARM core. NoInterrupt specifies that there is no interrupt. The address for the instruction code is in PC. In this theorem and all the following ones, I omitted some details, such as elements MEM_READ and MEM_WRITE that track what addresses are accessed in the execution. This information is not used in my work. Some detailed configuration information for the ARM core model is also omitted, including ARM ISA version, default settings and reserved bits in the CPSR register.

4.3.3 Hoare Logic

At the top layer, the ARM instruction semantics theorems described in Section 4.3.2 are lifted to Hoare triples in the program logic [60, 61]. To do so the system state is translated into a set. This set is then presented as the separation conjunction of the relevant elements from the system state. The separation between these elements is guaranteed because each element corresponds to some distinct hardware pieces on the processor, such as registers and memory. Now techniques such as precondition strengthening, postcondition weakening, frame rule, sequential composition, and branch combination can be used in proof. The readability of program specifications is also improved.

In the Cambridge ARM model, the Hoare triple for STRB r2, [r3] is shown in Figure 4.2. It is lifted from the theorem in Figure 4.1. Here, SPEC M P C Q models the Hoare triple $\{\ P\ \}\ \ C\ \{\ Q\ \}$ on machine M. The machine model M defines the next-state transition function among other things. For machine ARM_MODEL, ARM_NEXT NoInterrupt is the next-state transition function. The program is represented with the set of address-word pairs. $*$ is the separation conjunction operator used to connect the separation predicate. cond lifts the Boolean predicate into a separation logic assertion.

4.3.3.1 Stack Model

There is a stack model in the Cambridge ARM model. For every mode *amd*, the stack is abstracted into a list *xs*. The stack is specified relative to the frame pointer *fp*:

$$\begin{aligned}
&\mathsf{aSTACK}\ \mathit{amd}\ \mathit{fp}\ \mathit{xs} \triangleq\\
&\mathsf{case}\ \mathit{amd}\ \mathsf{of\ usr} \rightarrow \mathsf{aR}\ \mathsf{R}_{11}\ \mathit{fp} * \mathsf{aR}\ \mathsf{SPusr}\ (\mathit{fp} - (4 * \mathsf{LENGTH}\ \mathit{xs}))*\\
&\mathsf{SEP\_ARRAY}\ \mathsf{aM}\ (-4)\ \mathit{fp}\ \mathit{xs} * \mathsf{cond}\ (\mathsf{ALIGNED}\ \mathit{fp})\\
&\ldots
\end{aligned}$$

where SEP_ARRAY *mm stride start array* is an array abstraction from the Cambridge ARM model. It specifies that list *array* resides in a memory block starting from *start*. *mm* is the memory layout model. *stride* specifies the stride in addresses between adjacent memory cells, and its sign indicates if the addresses increases or decreases from the start position.

This stack model uses both the frame pointer register (R_{11}) and the stack pointer register (R_{13}). This design choice brings up some issues. First, R_{13} is not always reserved. For example, armcc only set it to the current stack frame pointer with option –use_frame_pointer. Second, automation is made harder because the length of the whole stack frame needs to be reasoned about.

4.3.3.2 Hiding as Weakening

For a given separation assertion, there are unlimited ways to weaken it. A special weakening operation is defined in the Cambridge ARM model. It is called hiding:

$$\neg p \triangleq \lambda\, s.\exists\, y.p\ y\ s.$$

In general, hiding weakens an assertion:

$$\vdash (\forall\ x.p\ x)\ \triangleright\ \neg p.$$

Here, $\triangleright$ is separation implication:[2]

$$p \triangleright q \triangleq \forall\, s.p\ s \Rightarrow q\ s.$$

For a Hoare triple, hiding the last parameter of the assertion in the postcondition weakens it:

$$\vdash \mathsf{SPEC}\, x\, p\ c\,(q\ *\ q'\ y)\ \Rightarrow\ \mathsf{SPEC}\, x\, p\ c\,(q\ * \neg q').$$

Hiding on the precondition only works when the parameter is universally qualified to the assertion. In other words, the variable to be hidden must not appear in the postcondition:

$$\vdash (\forall\, y.\ \mathsf{SPEC}\ x\,(p\ *\ p'\ y)\ c\ q)\quad\Longleftrightarrow\quad \mathsf{SPEC}\ x\,(p\ *\ \neg p')\ c\ q.$$

I use hiding as the main technique in weakening a Hoare triple, because it is easy to mechanize the process.

4.3.4 Automation

In the Cambridge ARM model there is an automation tool chain. It is used to automatically derive theorems for most instructions at the middle layer, as well as lift them into Hoare triples at the top layer. In the running example, theorems in Figures 4.1 and 4.2 are automatically proved. I extended this tool chain to support

[2]It is not the same in separation logic.

device access, stack operation, and operations on the current program status register (CPSR).

4.4 Extended ARM SoC Model

I extended the Cambridge ARM model to support the proof of programs with interrupts. I also introduced the notion of time in the program properties. Time is measured using the execution cycle of an ARM instruction. For simplicity, it is assumed all instructions have the same execution time. In some devices, the clock divider introduces a timer. Some interesting properties related to time can be expressed using this timer.

4.4.1 Device Model with Interrupts

The abstract model for devices with interrupts is defined in Figure 4.3. It extends the abstract device model in Section 2.2.1 with two elements. One is irqMap, which is the set of interrupts belonging to the device. The other is irqReq, which returns the active interrupt request from the device state.

4.4.2 ARM SoC Model

I extended the logic model of the ARM core into the logic model of an ARM SoC. The system state is expanded to include the state of the device. The device model is introduced into the system via three parameters:

1. Memory map, which tells if an address is mapped to devices or not.
2. The semantic model of the device, which describes the autonomous transition cycle of the device and how the device execute the commands from the processor core. The autonomous transition of the device and the possible side effects of the device access is now part of the execution cycle of the system.
3. ISR map, which is used to locate the address of the ISR upon an interrupt request.

I defined ARM_SOC_NEXT as the next-state transition function for the extended ARM SoC machine. In one cycle the ARM processor core is updated just as in the Cambridge ARM model, and the autonomous transition of the device is performed, too. If at the beginning of the cycle an interrupt request occurs, it will be handled after the execution of the current instruction. When the instruction is a load/store instruction, it is directed to the main memory or the devices by checking the address against the memory map. In the first case, the access is just like it is in the Cambridge ARM model. In the second case, if the address and size are mapped to the device, then the device is accessed, and the side effect of the device access on the device state is written back to the device state. Otherwise an exception is thrown. The memory read primitive for the ARM SoC model is defined in Figure 4.4. write_mem is extended in a similar way.

4.4.3 Atomic ARM Instruction Semantics

For any instruction, the interrupt handling process is the same. Given that deriving these theorems is time consuming, it would be better to isolate the interrupt handling part from the execution cycle ARM_SOC_NEXT with interrupt requests, derive its semantics once, and reuse it for all instructions.

4.4.3.1 Atomic ARM Machine

First, I defined the atomic next-state transition function ARM_ATOMIC_NEXT, which does not handle interrupt requests. To derive the semantics theorem for an instruction, we have to consider two cases, one without interrupt requests and the other with interrupt requests. The execution cycle ARM_SOC_NEXT with interrupt requests can be deconstructed into two parts:

$$
\begin{aligned}
\vdash\ & (\mathsf{uart0.irqReq}\ (\mathsf{ARM\_READ\_PSTATE}\ s) = \mathsf{SOME}\ \mathit{irpt}) \wedge \neg\mathsf{ARM\_READ\_UNDEF}\ s \\
& \wedge \neg\mathsf{ARM\_READ\_STATUS}\ \mathsf{PSR_I}\ s \wedge \\
& (\exists\, s_1.\mathsf{ARM\_ATOMIC\_NEXT}\ \mathsf{lpcMemMap}\ \mathsf{uart0}\ s = \mathsf{SOME}\ s_1) \Rightarrow \\
& (\mathsf{ARM\_SOC\_NEXT}\ \mathsf{lpcMemMap}\ \mathsf{uart0}\ \mathsf{lpcIsrMap}\ s = \\
& \mathsf{ARM\_ENTER\_IRQ}\ \mathsf{lpcIsrMap}\ \mathit{irpt} \\
& (\mathsf{THE}\ (\mathsf{ARM\_ATOMIC\_NEXT}\ \mathsf{lpcMemMap}\ \mathsf{uart0}\ s))).
\end{aligned}
\tag{4.2}
$$

The first part is the atomic execution ARM_ATOMIC_NEXT, similar to the case without interrupt requests. The second part is the handling of the interrupt ARM_ENTER_IRQ, which is the same for any instruction. It includes the context switch and the entry to the ISR.

The semantics of entry to the ISR are described in the theorem in Figure 4.5. When the interrupt *irpt* is taken, a context switch happens. CPSR is saved in SPSR of the irq mode. Interrupt handling then is disabled by masking the irq bit in the CPSR. *isrMap irpt* returns the entry address of the ISR for handling *irpt*. Note that the entry point for the hardware interrupt handling is hard-coded as the start address of the ISR since the interrupt vectoring is not considered here for simplicity.

4.4.3.2 Atomic Instruction Semantics for SoC

ARM processor has a feature called register banking. Basically, there are multiple physical copies for some registers. For example, each mode has its own physical link register. After a mode switch, link register is physically another one.

In the Cambridge ARM model, mode transition through modifying CPSR is not supported except at the bottom layer. This design choice results in much cleaner theorems. For example, the details of register banking in different modes are not visible.

However, interrupt handling involves the processor mode switch. So I made register banks across different modes explicit at the middle and top layers.

Since the device is introduced into the model, there are two theorems describing the semantics for each load/store instruction. One is for the case when memory is accessed. The other is for the case when the device is accessed.

In my extended model, there are two theorems, shown in Figure 4.6 and Figure 4.7, for instruction STRB r2, [r3]. The theorem in Figure 4.6 describes the memory access. Compared to the theorem in Equation 4.1 there are some changes. There are two new parameters: *addrMapped* and *dev*. $\neg$*addrMapped* x means that x is not mapped to any device. *dev* describes the behavior of the device. In the example, addresses in PC and register 3 are assumed to be in the memory. *dev*.transit (ARM_READ_PSTATE s) is the next device state as the result of the autonomous transition. After the execution cycle, the device state is updated by ARM_WRITE_PSTATE. Register banking in different modes are exposed; Register access function has one extra parameter describing the mode. In this example, ARM_READ_REG_MODE $(15, 19)$ s reads register 15 in mode 19 (supervisor, or svc mode).

The theorem in Figure 4.7 describes the device access. In contrast to the theorem in Figure 4.6, it is assumed here that the address in register 3 is indeed mapped to the device. Furthermore, it is assumed that the device access would not cause exceptions. These two additional assumptions need to be proved in composition. In addition to the autonomous transition, the side effect of the device access is also reflected in the next device state.

4.4.4 Single-Step Hoare Logic for ARM SoC

Deconstructing the instruction cycle with interrupt requests into two parts calls for a single-step Hoare logic for the ARM SoC model, because the execution of the instruction must be finished before interrupt handling begins. In this section I introduce a single-step Hoare logic for the ARM SoC model. It is used to bridge the gap between the atomic ARM SoC model and interrupt-driven ARM SoC model.

4.4.4.1 Single-Step Hoare Logic

At the top layer, a single-step Hoare logic is needed to express the deconstruction of the execution cycle with interrupt requests. STEP defines a single-step total-correctness Hoare logic:

$$\begin{aligned}&\mathsf{STEP}\ (\mathit{to\_set}, \mathit{next}, \mathit{instr})\ p\ q \triangleq\\ &\forall\, s\ r.(p * r)\ (\mathit{to\_set}\ s) \Rightarrow (\exists\, x.\mathit{next}\ s\ x) \wedge \forall\, y.\mathit{next}\ s\ y \Rightarrow (q * r)\ (\mathit{to\_set}\ y).\end{aligned}$$

Here, $(\mathit{to\_set}, \mathit{next}, \mathit{instr})$ defines a machine model. *to_set* translates the system state of the machine to a set. *next* is the next-state transition function. *instr* is used to lift instruction code from the memory.

After the instruction code is lifted, we have a Hoare triple definition INS_SPEC:

$$\begin{aligned}&\mathsf{INS\_SPEC}\ (\mathit{to\_set}, \mathit{next}, \mathit{instr})\ p\ c\ q\ \triangleq\\ &\mathsf{STEP}\ (\mathit{to\_set}, \mathit{next}, \mathit{instr})\ (\mathsf{CODE\_POOL}\ \mathit{instr}\ c\ *\ p)\\ &(\mathsf{CODE\_POOL}\ \mathit{instr}\ c * q).\end{aligned}$$

Here, CODE_POOL is syntactic sugar. The instruction code c is specified in the same memory with data using separation conjunction. The single-step Hoare logic

INS_SPEC is a proper subset of the program logic SPEC:

$$\vdash \mathsf{INS\_SPEC}\ m\ p\ c\ q \Rightarrow \mathsf{SPEC}\ m\ p\ c\ q. \tag{4.3}$$

INS_SPEC supports most of the proof rules of SPEC except the sequential composition. Since INS_SPEC is used for deriving instruction semantics only, this is not an issue for me.

4.4.4.2 Single-Step Hoare Logic for ARM SoC

Hoare triples for memory access and device access scenarios for instruction STRB r2, [r3] are shown, respectively, in Figures 4.8 and 4.9. They are lifted from the theorems in Figures 4.6 and 4.7.

Machine LPC_MODEL has ARM_ATOMIC_NEXT as the next-state transition function. So it ignores the interrupt requests. aBYTE_MEMORY *df f* specifies a byte-level memory block. Its domain is *df*, and *f* maps an address to a byte. The assumption that addresses in *df* are not mapped to devices is already built in the definition of aBYTE_MEMORY *df f*.

4.5 Hoare Logic for ARM SoC Model

I have extended the Cambridge ARM model to an ARM SoC model. However, the semantics do not support interrupt handling, and the Hoare logic does not support sequential composition, which is the fundamental technique in Hoare logic. Actually, the single-step atomic semantics of the ARM SoC model can be easily extended to meet these two crucial requirements.

4.5.1 Reintroduce Interrupts

To introduce interrupts back to the semantics of the ARM instructions, I rely on theorems in Figures 4.10 and 4.11. The theorems are in the form of $ht_{\text{atomic}} \Rightarrow ht_{\text{irq}}$. ht_{atomic} denotes the Hoare triple for the single-step atomic semantics of an ARM instruction defined over machine LPC_MODEL with ARM_ATOMIC_NEXT as the next-state transition function. This machine does not support interrupt handling. ht_{irq} denotes the Hoare triple for the single-step semantics of an ARM instruction, which is defined over machine LPC_IRQ_MODEL with ARM_SOC_NEXT as the next-state transition function. This machine supports interrupt handling.

These two theorems are proved using the theorem in Equation 4.2. They are used to rewrite the single-step semantics for an ARM instruction from the atomic ARM SoC machine to the ARM SoC machine. The former is used when there are no interrupt requests or the interrupt bit is masked, while the latter is used when there are interrupt requests and the interrupt bit is not masked.

More specifically, the theorem in Figure 4.10 states that if the Hoare triple for the single-step atomic semantics of an ARM instruction is known, the Hoare triple for the single-step semantics without interrupts pending can be derived by strengthening the precondition with the assertion that there are no interrupted pending. The theorem in Figure 4.11 states that if the Hoare triple for the single-step atomic semantics of an ARM instruction is known, the Hoare triple for the single-step semantics with interrupts pending can be derived by adding the separation assertions reflecting the context switch. $\mathsf{aPSR\ ips}\ x$ is used to describe the program status register (PSR) in the interrupt mode. PSR is used to back up the CPSR in the previous mode *m1* in the context switch in interrupt handling.

The fields in CPSR such as the mode ($\mathsf{aMD\ \ x}$), the status bits ($\mathsf{aS1\ PSR_x}$), and other fields (aCPSR) are explicitly specified in the precondition and postconditions of the Hoare triples to accommodate MSR instruction, which is used to implement a context switch by modifying CPSR. For other instructions, fields in CPSR should be the same in the precondition and postcondition for a Hoare triple. In fact, these fields of CPSR will not appear because of the frame rule.

The single-step atomic semantics of the ARM instructions are intended to be expanded with the high-level, application specific properties, and to be combined with the theorem describing the effect of the ISR later on in the context of the application. I adapted the automation tool from the Cambridge ARM model to derive the theorems of the single-step atomic semantics of the ARM instructions automatically.

Using the theorem in Equation 4.3 we can further extend the theorems into SPEC assertions, which support sequential composition. With my approach, the extension of the Cambridge model is efficient. Repetition in proof is reduced, and we can derive the instruction semantics theorems with the effect of ISR included, from a relatively small foundation.

4.5.2 Inference Rules

In this section I summarize some high-level inference rules to be used to facilitate the proof. Application of these rules cannot always be automated.

4.5.2.1 Integrate the Effect of the ISR

When time is not considered, delay in the execution is not a concern. The effect of the ISR can always be weakened to maintaining an invariant. The rule in Figure 4.12 describes how to integrate the effect of the ISR into an instruction semantics theorem. Here, p_1 and q_1 are separation assertions over resource r_1. p_2, q_2 and iv_1 are separation assertions over resource r_2. iv_2 is a separation assertion over resource r_3. r_1, r_2, r_3 are separated.

iv_2 is always maintained by the instruction. The extended semantics are obtained by weakening q_2 to iv_1, adding invariant iv_2 over r_3 to the semantics of the instruction, and the atomic machine LPC_MODEL is replaced with the interruptible machine LPC_IRQ_MODEL.

With time considered, it gets more complicated, because it is not always possible to weaken the Hoare triple of the ISR to maintaining an invariant. The rule in Figure 4.13 describes how to integrate the effect of the ISR into an instruction semantics theorem. Here, p_1 and q_1 are separation assertions over resource r_1. p_2, q_2 and ip_1 are separation assertions over resource r_2. iv_2 is a separation assertion over resource r_3. r_1, r_2, r_3 are separated.

iv_2 is always maintained by the instruction. The extended semantics are obtained by weakening q_2 and iq_2 to iv_1, adding invariant iv_2 over r_3 to the semantics of the instruction, and the atomic machine LPC_MODEL is replaced with the interruptible machine LPC_IRQ_MODEL.

Each rule is actually a combination of postcondition weakening, sequential composition, and branch combination.

4.5.2.2 Introduce High-Level Assertions

A high-level assertion is usually introduced by grouping the related separation conjunctions first into a single assertion and then strengthening it with a predicate via cond. The rule in Figure 4.14 does this. Suppose g is the predicate that needs to be introduced via cond, and is defined over a group of elements in the system state. These elements have value s_1 in the precondition, and s_2 in the postcondition. To introduce $\mathsf{cond}\ (g\ x\ )$, the only proof obligation is to show $g\ s_1 \Rightarrow g\ s_2$.

4.5.3 Stack Model

With Hoare logic we can use high-level abstraction to specify properties. When reasoning about an assembly program, stack abstraction is useful. For example, stack manipulation is an important part of a context switch. Two specific ARM instructions, push and pop, perform stack operations.

I designed a partial stack model using the stack pointer register (register 13 in ARM mode). It specifies the used stack space and the reserved space for the stack to grow. The composition function will automatically match this specification across two instructions by adding more reserved space or revealing more stack if necessary.

The model is partial because the frame pointer specification is omitted for simplicity. It can be added for a complete stack specification at the cost of increasing complexity in automation. Soundness of the stack specification is not at risk since frame rule can be used to augment the frame pointer specification.

The stack in the usr mode is defined below and is part of the aPSTACK definition:

$$\begin{aligned}&\mathsf{aPSTACK}\ amd\ sp\ stk\ resv \triangleq\\&\mathsf{case}\ amd\ \mathsf{of\ usr}\ \rightarrow \mathsf{aR\ SPusr}\ sp * \mathsf{SEP\_ARRAY\ aM}\ (-4)(sp-4)resv *\\&\mathsf{SEP\_ARRAY\ aM}\ 4\ sp\ stk * \mathsf{cond}\ (\mathsf{ALIGNED}\ sp)\\&\ldots\end{aligned}$$

When describing a safe stack operation, the length, rather than the content in the reserved space is important. The following definition can be used:

$$\begin{aligned}&\mathsf{aPPSTACK}\ amd\ sp\ stk\ resvsize \triangleq\\&(\lambda\, s.\exists\, y.(\mathsf{aPSTACK}\ amd\ sp\ stk\ y\ * \mathsf{cond}\ (\mathsf{LENGTH}\ y = resvsize))s)\end{aligned}$$

where *resvsize* denotes the size of the reserved space. When it is used to assert the precondition and postcondition of a clean context switch for a function call or interrupt handling, it will take the shape of a nice invariant.

4.5.3.1 Automation of Stack Assertion

Stack assertion is automated in deriving the semantics for stack operation instructions and composing sequential instructions. In ARM architecture, the convention is that the stack grows downwards. The stack pointer points to the top item on the stack. This is used to identify the stack operations. Stack is introduced for individual instructions abstractly. For push instructions, stack grows from empty to some length. For pop instructions, the stack shrinks to empty. This makes the task easier.

The algorithm starts by looking for aR SPx *sp* assertions. Here, SPx indicates the stack pointer for mode x. If no such assertions exist, the instruction is not a stack operation. Otherwise, the stack pointer *sp* is extracted. The algorithm proceeds by looking for assertions with pattern aM *a y*, which asserts that byte *y* is stored at memory address *a*. If nothing is found, the instruction is not a stack operation. Otherwise, the algorithm tries to find if the stack pointer is in the addresses in the assertions found in the last step. If yes, this is a pop instruction; otherwise, it is a push instruction.

Once the stack operation is identified, the lists for the stack or the reserved area are built. The stack assertion is introduced using special forms of the definition when either the stack or the reserved list is empty.

Stacks are matched in sequential composition. The first step is to match the stack pointer. Then the length for the list for the stack or the reserved area is matched by extending the short ones. In the last step, the variables are unified.

This automation is designed for push and pop only. For instruction with random access on the stack based on the stack pointer, the stack assertion can not be automatically introduced, but it can be done. For random stack access using the stack frame pointer, the definition needs to be expanded with the frame pointer register.

$$
\begin{array}{l}
\vdash (\mathsf{ARM\_MODE}\ s = 19) \wedge \\
\quad \mathsf{aligned}\ \ (\mathsf{ARM\_READ\_REG}\ 15\ s, 4) \wedge \\
\quad (\mathsf{ARM\_READ\_MEM}\ (\mathsf{ARM\_READ\_REG}\ 15\ s + 3)\ s = 229) \wedge \\
\quad (\mathsf{ARM\_READ\_MEM}\ (\mathsf{ARM\_READ\_REG}\ 15\ s + 2)\ s = 195) \wedge \\
\quad (\mathsf{ARM\_READ\_MEM}\ (\mathsf{ARM\_READ\_REG}\ 15\ s + 1)\ s = 32) \wedge \\
\quad (\mathsf{ARM\_READ\_MEM}\ (\mathsf{ARM\_READ\_REG}\ 15\ s)\ s = 0) \Rightarrow \\
\quad (\mathsf{ARM\_NEXT}\ \mathsf{NoInterrupt}\ s = \\
\quad \mathsf{SOME}\ (\mathsf{CLEAR\_EXCLUSIVE\_BY\_ADDRESS}\ (\mathsf{ARM\_READ\_REG}\ 3\ s, 1) \\
\quad (\mathsf{ARM\_WRITE\_MEM}\ (\mathsf{ARM\_READ\_REG}\ 3\ s) \\
\qquad\qquad ((\mathsf{ARM\_READ\_REG}\ 2\ s) \uparrow^{8}_{32}) \\
\quad (\mathsf{ARM\_WRITE\_REG}\ 15\ (\mathsf{ARM\_READ\_REG}\ 15\ s + 4)\ s))))
\end{array}
$$

Figure 4.1: Semantics of instruction STRB r2, [r3] in the Cambridge ARM model

$$
\begin{array}{l}
\vdash \mathsf{SPEC\ ARM\_MODEL} \\
\quad (\mathsf{aPC}\ p * \mathsf{aR}\ 3\ r_3 * \mathsf{aR}\ 2\ r_2 * \mathsf{aBYTE\_MEMORY}\ df\, f\ *\ \mathsf{cond}(r_3 \in df)) \\
\quad \{\ (p, 0xE5C32000)\ \} \\
\quad (\mathsf{aPC}\ (p+4) * \mathsf{aR}\ 3\ r_3 * \mathsf{aR}\ 2\ r_2 * \mathsf{aBYTE\_MEMORY}\ df\, ((r_3 \mapsto r_2 \uparrow^{8}_{32})\ f))
\end{array}
$$

Figure 4.2: Hoare triple for instruction STRB r2, [r3] in the Cambridge ARM model

$$
\begin{array}{l}
<\!|\,\mathsf{mapped} : \mathsf{addr} \rightarrow \mathsf{bool};\\
\quad \mathsf{mapped\_read} : \mathsf{addr} \rightarrow \tau \rightarrow \mathsf{word} * \ \mathsf{bool} * \tau;\\
\quad \mathsf{mapped\_write} : \mathsf{addr} \rightarrow \mathsf{data} \rightarrow \tau \rightarrow \mathsf{bool} * \tau;\\
\quad \mathsf{transit} : \tau \rightarrow \tau;\\
\quad \mathsf{irqMap} : irq \rightarrow bool;\\
\quad \mathsf{irqReq} : \tau \rightarrow irqoption;\\
\quad \mathsf{wellform} : \tau \rightarrow \mathsf{bool}.\,|\!>
\end{array}
$$

Figure 4.3: Abstract model for devices with interrupts

```
read_mem $addrMapped\ dev\ ii\ (memaddrdesc, size) \triangleq$ if$size \notin \{1; 2; 4; 8\}$
    then errorT "read_mem: size is not 1, 2, 4 or 8"
    else (let $address = memaddrdesc.$paddress in
            if isMemAccess $addrMapped\ address\ size$
            then forT $0\ (size - 1)(\lambda\, i.$ read_mem1 $ii\ (address +$ n2 $i))$
            else if isDevAccess $addrMapped\ address\ size\ \wedge$
                    $(dev.$addr_map $address = size)$
                then read_io $dev\ ii\ address$
                else errorT "dev read: not defined")
```

Figure 4.4: Memory read primitive for the ARM SoC model

$$
\begin{array}{l}
\vdash \textsf{GOOD\_MODE}\ (\textsf{ARM\_MODE}\ s) \Rightarrow \\
\quad (\textsf{ARM\_ENTER\_IRQ}\ \mathit{isrMap}\ \mathit{irpt}\ s = \\
\quad \textsf{SOME}\ (\textsf{ARM\_WRITE\_IT}\ 0\ (\textsf{ARM\_WRITE\_STATUS}\ \textsf{PSR}_{\mathrm{A}}\ \textsf{T} \\
\quad (\textsf{ARM\_WRITE\_STATUS}\ \textsf{PSR}_{\mathrm{I}}\ \textsf{T}\ (\textsf{ARM\_WRITE\_MODE}\ 18 \\
\quad (\textsf{ARM\_WRITE\_SPSR\_MODE}\ 18\ (\textsf{ARM\_READ\_CPSR}\ s) \\
\quad (\textsf{ARM\_WRITE\_REG\_MODE}\ (15,16)\ (\mathit{isrMap}\ \mathit{irpt}) \\
\quad (\textsf{ARM\_WRITE\_REG\_MODE}\ (14,18)\ s))))))))
\end{array}
$$

Figure 4.5: Semantics for ARM_ENTER_IRQ

$$
\begin{array}{l}
\vdash (\mathsf{ARM\_MODE}\ s = 19) \land \mathsf{aligned}\ (\mathsf{ARM\_READ\_REG\_MODE}\ (15,19)\ s, 4) \land \\
\quad (\mathsf{ARM\_READ\_MEM}\ (\mathsf{ARM\_READ\_REG\_MODE}\ (15,19)\ s+3)\ s = 22) \land \\
\quad (\mathsf{ARM\_READ\_MEM}\ (\mathsf{ARM\_READ\_REG\_MODE}\ (15,19)\ s+2)\ s = 195) \land \\
\quad (\mathsf{ARM\_READ\_MEM}\ (\mathsf{ARM\_READ\_REG\_MODE}\ (15,19)\ s+1)\ s = 32) \land \\
\quad (\mathsf{ARM\_READ\_MEM}\ (\mathsf{ARM\_READ\_REG\_MODE}\ (15,19)\ s)\ s = 0) \land \\
\quad \neg \mathit{addrMapped}\ (\mathsf{ARM\_READ\_REG\_MODE}\ (3,19)\ s) \land \\
\quad \neg \mathit{addrMapped}\ (\mathsf{ARM\_READ\_REG\_MODE}\ (3,19)\ s) \land \\
\quad \neg \mathit{addrMapped}\ (\mathsf{ARM\_READ\_REG\_MODE}\ (15,19)\ s) \land \\
\quad \neg \mathit{addrMapped}\ (\mathsf{ARM\_READ\_REG\_MODE}\ (15,19)\ s) \land \\
\quad \neg \mathit{addrMapped}\ (\mathsf{ARM\_READ\_REG\_MODE}\ (15,19)\ s+1) \land \\
\quad \neg \mathit{addrMapped}\ (\mathsf{ARM\_READ\_REG\_MODE}\ (15,19)\ s+2) \land \\
\quad \neg \mathit{addrMapped}\ (\mathsf{ARM\_READ\_REG\_MODE}\ (15,19)\ s+3) \Rightarrow \\
\quad (\mathsf{ARM\_ATOMIC\_NEXT}\ \mathit{addrMapped}\ \mathit{dev}\ s = \\
\quad \mathsf{SOME}\ (\mathsf{CLEAR\_EXCLUSIVE\_BY\_ADDRESS} \\
\qquad\qquad (\mathsf{ARM\_READ\_REG\_MODE}\ (3,19)\ s, 1) \\
\quad (\mathsf{ARM\_WRITE\_PSTATE}\ (\mathit{dev}.\mathsf{transit}\ (\mathsf{ARM\_READ\_PSTATE}\ s)) \\
\quad (\mathsf{ARM\_WRITE\_MEM}\ (\mathsf{ARM\_READ\_REG\_MODE}\ (3,19)\ s) \\
\qquad\qquad ((\mathsf{ARM\_READ\_REG\_MODE}\ (2,19)\ s) \uparrow^{8}_{32}) \\
\quad (\mathsf{ARM\_WRITE\_REG\_MODE}\ (15,19) \\
\quad (\mathsf{ARM\_READ\_REG\_MODE}\ (15,19)\ s+4)\ s)))))
\end{array}
$$

Figure 4.6: Memory access scenario for instruction STRB r2, [r3] in the ARM SoC model

$$
\begin{array}{l}
\vdash (\mathsf{ARM\_MODE}\ s = 19) \wedge \mathsf{aligned}\ (\mathsf{ARM\_READ\_REG\_MODE}\ (15, 19)\ s, 4) \wedge \\
\quad (\mathsf{ARM\_READ\_MEM}\ (\mathsf{ARM\_READ\_REG\_MODE}\ (15, 19)\ s + 3)\ s = 22) \wedge \\
\quad (\mathsf{ARM\_READ\_MEM}\ (\mathsf{ARM\_READ\_REG\_MODE}\ (15, 19)\ s + 2)\ s = 195) \wedge \\
\quad (\mathsf{ARM\_READ\_MEM}\ (\mathsf{ARM\_READ\_REG\_MODE}\ (15, 19)\ s + 1)\ s = 32) \wedge \\
\quad (\mathsf{ARM\_READ\_MEM}\ (\mathsf{ARM\_READ\_REG\_MODE}\ (15, 19)\ s)\ s = 0) \wedge \\
\quad \neg\mathsf{FST}(\mathit{dev}.\mathsf{mapped\_write}\ (\mathsf{ARM\_READ\_REG\_MODE}\ (3, 19)\ s) \\
\qquad\qquad [(\mathsf{ARM\_READ\_REG\_MODE}\ (2, 19)\ s) \uparrow^{8}_{32}] \\
\qquad\qquad (\mathsf{ARM\_READ\_PSTATE}\ s)) \wedge \\
\quad \mathit{addrMapped}\ (\mathsf{ARM\_READ\_REG\_MODE}\ (3, 19)\ s) \wedge \\
\quad \mathit{addrMapped}\ (\mathsf{ARM\_READ\_REG\_MODE}\ (3, 19)\ s) \wedge \\
\quad (\mathit{dev}.\mathsf{addr\_map}(\mathsf{ARM\_READ\_REG\_MODE}\ (3, 19)\ s) = 1) \wedge \\
\quad \neg\mathit{addrMapped}\ (\mathsf{ARM\_READ\_REG\_MODE}\ (15, 19)\ s) \wedge \\
\quad \neg\mathit{addrMapped}\ (\mathsf{ARM\_READ\_REG\_MODE}\ (15, 19)\ s) \wedge \\
\quad \neg\mathit{addrMapped}\ (\mathsf{ARM\_READ\_REG\_MODE}\ (15, 19)\ s + 1) \wedge \\
\quad \neg\mathit{addrMapped}\ (\mathsf{ARM\_READ\_REG\_MODE}\ (15, 19)\ s + 2) \wedge \\
\quad \neg\mathit{addrMapped}\ (\mathsf{ARM\_READ\_REG\_MODE}\ (15, 19)\ s + 3) \Rightarrow \\
\quad (\mathsf{ARM\_ATOMIC\_NEXT}\mathit{addrMapped}\ \mathit{dev}\ s = \\
\quad \mathsf{SOME}\ (\mathsf{CLEAR\_EXCLUSIVE\_BY\_ADDRESS} \\
\qquad\qquad (\mathsf{ARM\_READ\_REG\_MODE}\ (3, 19)\ s, 1) \\
\quad (\mathsf{ARM\_WRITE\_PSTATE} \\
\qquad\qquad (\mathit{dev}.\mathsf{transit}\ (\mathsf{SND}\ (\mathit{dev}.\mathsf{mapped\_write} \\
\qquad\qquad (\mathsf{ARM\_READ\_REG\_MODE}\ (3, 19)\ s) \\
\qquad\qquad [(\mathsf{ARM\_READ\_REG\_MODE}\ (2, 19)\ \ s) \uparrow^{8}_{32}] \\
\qquad\qquad (\mathsf{ARM\_READ\_PSTATE}\ s)))) \\
\quad (\mathsf{ARM\_WRITE\_REG\_MODE}\ (15, 19) \\
\quad (\mathsf{ARM\_READ\_REG\_MODE}\ (15, 19)\ s + 4)\ s))))
\end{array}
$$

Figure 4.7: Device access scenario for instruction STRB r2, [r3] in the ARM SoC model

$$
\begin{array}{l}
\vdash \mathsf{INS\_SPEC\ LPC\_MODEL} \\
\quad (\mathsf{aP}\ ps\ *\ \mathsf{aMD}\ 19\ *\ \mathsf{aPC}\ p\ *\ \mathsf{aR\ R3}\ r_3\ *\ \mathsf{aR\ R2}\ r_2\ *\mathsf{aBYTE\_MEMORY}\ df\,f\ * \\
\quad \mathsf{cond}(\neg\mathsf{lpcMemMap}\ p\ \wedge\ \neg\mathsf{lpcMemMap}\ (p\ +\ 1)\ \wedge \\
\quad \neg\mathsf{lpcMemMap}\ (p\ +\ 2)\ \wedge\neg\mathsf{lpcMemMap}\ (p\ +\ 3)\ \wedge\ r_3\ \in\ df)) \\
\quad \{\ (p, 0xE5C32000)\ \} \\
\quad (\mathsf{aP}\ (\mathsf{uart0.transit}\ ps)\ *\ \mathsf{aMD}\ 19\ *\ \mathsf{aPC}\ (p\ +\ 4)\ *\ \mathsf{aR\ R3}\ r_3\ *\mathsf{aR\ R2}\ r_2\ * \\
\quad \mathsf{aBYTE\_MEMORY}\ df\,((r_3\ \mapsto\ r_2\uparrow^{8}_{32})\ f))
\end{array}
$$

Figure 4.8: Semantics of instruction STRB r2, [r3] in the case of memory access in the single-step Hoare logic

$$
\begin{array}{l}
\vdash \mathsf{INS\_SPEC\ LPC\_MODEL} \\
\quad (\mathsf{aP}\ ps\ *\ \mathsf{aMD}\ 19\ *\ \mathsf{aPC}\ p\ *\ \mathsf{aR\ R3}\ r_3\ *\ \mathsf{aR\ R2}\ r_2\ * \\
\quad \mathsf{cond}(\neg\mathsf{FST}\ (\mathsf{uart0.mapped\_write}\ r_3\ [r_2 \uparrow^{8}_{32}]\ ps)\ \wedge\ \mathsf{lpcMemMap}\ r_3\ \wedge \\
\quad (\mathsf{uart0.addr\_map}\ r_3\ =\ 1)\ \wedge\ \neg\mathsf{lpcMemMap}\ p\ \wedge\ \neg\mathsf{lpcMemMap}\ (p\ +\ 1)\ \wedge \\
\quad \neg\mathsf{lpcMemMap}\ (p\ +\ 2)\ \wedge\ \neg\mathsf{lpcMemMap}\ (p\ +\ 3))) \\
\quad \{\ (p, 0xE5C32000)\ \} \\
\quad (\mathsf{aP}\ (\mathsf{uart0.transit}\ (\mathsf{SND}\ (\mathsf{uart0.mapped\_write}\ r_3\ [r_2 \uparrow^{8}_{32}]\ ps)))\ *\ \mathsf{aMD}\ 19\ * \\
\quad \mathsf{aPC}\ (p\ +\ 4)\ *\ \mathsf{aR\ R3}\ r_3\ *\ \mathsf{aR\ R2}\ r_2)
\end{array}
$$

Figure 4.9: Semantics of instruction STRB r2, [r3] in the case of device access in the single-step Hoare logic

$$\begin{array}{l}
\vdash \mathsf{INS\_SPEC\ LPC\_MODEL} \\
\quad (p * \mathsf{aPC}\ pc_1 * \mathsf{aCPSR}\ cpsr_1 * \mathsf{aS1\ PSR}_{\mathrm{N}}\ psrn_1 * \mathsf{aS1\ PSR}_{\mathrm{Z}}\ psrz_1 * \\
\quad \mathsf{aS1\ PSR}_{\mathrm{C}} psrc_1 * \mathsf{aS1\ PSR}_{\mathrm{V}}\ psrv_1 * \mathsf{aS1\ PSR}_{\mathrm{Q}}\ psrq_1 * \\
\quad \mathsf{aS1\ PSR}_{\mathrm{A}}\ psra_1 * \mathsf{aS1\ PSR}_{\mathrm{I}}\ psri_1 * \mathsf{aP}\ ps_1 * \mathsf{aMD}\ m_1) \\
\quad c \\
\quad (q * \mathsf{aPC}\ pc_2 * \mathsf{aCPSR}\ cpsr_2 * \mathsf{aS1\ PSR}_{\mathrm{N}}\ psrn_2 * \mathsf{aS1\ PSR}_{\mathrm{Z}}\ psrz_2 * \\
\quad \mathsf{aS1\ PSR}_{\mathrm{C}}\ psrc_2 * \mathsf{aS1\ PSR}_{\mathrm{V}} psrv_2 * \mathsf{aS1\ PSR}_{\mathrm{Q}}\ psrq_2 * \\
\quad \mathsf{aS1\ PSR}_{\mathrm{A}}\ psra_2 * \mathsf{aS1\ PSR}_{\mathrm{I}}\ psri_2 * \mathsf{aP}\ ps_2 * \mathsf{aMD}\ m_2) \\
\quad \Rightarrow \\
\quad \mathsf{INS\_SPEC\ LPC\_IRQ\_MODEL} \\
\quad (p * \mathsf{aPC}\ pc_1 * \mathsf{aCPSR}\ cpsr_1 * \mathsf{aS1\ PSR}_{\mathrm{N}}\ psrn_1 * \mathsf{aS1\ PSR}_{\mathrm{Z}}\ psrz_1 * \\
\quad \mathsf{aS1\ PSR}_{\mathrm{C}}\ psrc_1 * \mathsf{aS1\ PSR}_{\mathrm{V}}\ psrv_1 * \mathsf{aS1\ PSR}_{\mathrm{Q}}\ psrq_1 * \\
\quad \mathsf{aS1\ PSR}_{\mathrm{A}}\ psra_1 * \mathsf{aS1\ PSR}_{\mathrm{I}}\ psri_1 * \mathsf{aP}\ ps_1 * \mathsf{aMD}\ m_1 * \\
\quad \mathsf{precond}(psri_1 \vee (\mathsf{uart0.irqReq}\ ps_1 = \mathsf{NONE}))) \\
\quad c \\
\quad (q * \mathsf{aPC}\ pc_2 * \mathsf{aCPSR}\ cpsr_2 * \mathsf{aS1\ PSR}_{\mathrm{N}}\ psrn_2 * \mathsf{aS1\ PSR}_{\mathrm{Z}}\ psrz_2 * \\
\quad \mathsf{aS1\ PSR}_{\mathrm{C}}\ psrc_2 * \mathsf{aS1\ PSR}_{\mathrm{V}}\ psrv_2 * \mathsf{aS1\ PSR}_{\mathrm{Q}}\ psrq_2 * \\
\quad \mathsf{aS1\ PSR}_{\mathrm{A}}\ psra_2 * \mathsf{aS1\ PSR}_{\mathrm{I}}\ psri_2 * \mathsf{aP}\ ps_2 * \mathsf{aMD}\ m_2)
\end{array}$$

Figure 4.10: One can derive the single-step semantics without interrupts pending from the single-step atomic semantics for an ARM instruction

$$
\begin{array}{l}
\vdash \mathsf{INS\_SPEC\ LPC\_MODEL} \\
\quad (p * \mathsf{aPC}\ pc_1 * \mathsf{aCPSR}\ cpsr_1 * \mathsf{aS1\ PSR_N}\ psrn_1 * \mathsf{aS1\ PSR_Z}\ psrz_1 * \\
\quad \mathsf{aS1\ PSR_C}\ psrc_1 * \mathsf{aS1\ PSR_V}\ psrv_1 * \mathsf{aS1\ PSR_Q}\ psrq_1 * \\
\quad \mathsf{aS1\ PSR_A}\ psra_1 * \mathsf{aS1\ PSR_I} psri_1 * \mathsf{aP}\ ps_1 * \mathsf{aMD}\ m_1) \\
\quad c \\
\quad (q * \mathsf{aPC}\ pc_2 * \mathsf{aCPSR}\ cpsr_2 * \mathsf{aS1\ PSR_N}\ psrn_2 * \mathsf{aS1\ PSR_Z}\ psrz_2 * \\
\quad \mathsf{aS1\ PSR_C}\ psrc_2 * \mathsf{aS1\ PSR_V}\ psrv_2 * \mathsf{aS1\ PSR_Q} psrq_2 * \\
\quad \mathsf{aS1\ PSR_A}\ psra_2 * \mathsf{aS1\ PSR_I}\ psri_2 * \mathsf{aP}\ ps_2 * \mathsf{aMD}\ m_2) \\
\quad \Rightarrow \\
\quad \mathsf{INS\_SPEC\ LPC\_IRQ\_MODEL} \\
\quad (\mathsf{aPC}\ pc_1 * \mathsf{aCPSR}\ cpsr_1 * \mathsf{aR\ LRirq}\ lrirq * \mathsf{aPSR\ ips}\ ipsr * \\
\quad \mathsf{aS1\ PSR_N}\ psrn_1 * \mathsf{aS1\ PSR_Z}\ psrz_1 * \mathsf{aS1\ PSR_C}\ psrc_1 * \\
\quad \mathsf{aS1\ PSR_V}\ psrv_1 * \mathsf{aS1\ PSR_Q}\ psrq_1 * \mathsf{aS1\ PSR_A}\ psra_1 * \\
\quad \mathsf{aS1\ PSR_I} psri_1 * \mathsf{aP}\ ps_1 * \mathsf{aMD}\ m_1 * \\
\quad \mathsf{cond}(\mathsf{lpcIsrMap}(\mathsf{THE}(\mathsf{uart0.irqReq}\ ps_1))\&\&3 = 0) * \\
\quad \mathsf{precond}\ (\neg psri_1 \wedge \mathsf{IS\_SOME}\ (\mathsf{uart0.irqReq}\ ps_1)) * p) \\
\quad c \\
\quad (\mathsf{aPC}\ (\mathsf{lpcIsrMap}\ (\mathsf{THE}\ (\mathsf{uart0.irqReq}\ ps_1))) * \mathsf{aCPSR}\ cpsr_2 * \\
\quad \mathsf{aR\ LRirq}\ (pc_2 + 4) * \\
\quad \mathsf{aPSR\ ips} <| \mathsf{N} := psrn_2; \mathsf{Z} := psrz_2; \mathsf{C} := psrc_2; \mathsf{V} := psrv_2; \mathsf{Q} := psrq_2; \\
\quad \mathsf{IT} := 0; \mathsf{J} := \mathsf{F}; \mathsf{Reserved} := cpsr_2.\mathsf{res}; \mathsf{GE} := cpsr_2.\mathsf{ge}; \\
\quad \mathsf{E} := \mathsf{F}; \mathsf{A} := psra_2; \mathsf{I} := psri_2; \mathsf{F} := \mathsf{T}; \mathsf{T} := \mathsf{F}; \mathsf{M} := m_2 |> * \\
\quad \mathsf{aS1\ PSR_N}\ psrn_2 * \mathsf{aS1\ PSR_Z}\ psrz_2 * \mathsf{aS1\ PSR_C}\ psrc_2 * \mathsf{aS1\ PSR_V} psrv_2 * \\
\quad \mathsf{aS1\ PSR_Q}\ psrq_2 * \mathsf{aS1\ PSR_A}\ \mathsf{T} * \mathsf{aS1\ PSR_I}\ \mathsf{T} * \mathsf{aP}\ ps_2 * \mathsf{aMD}\ 18\ * q)
\end{array}
$$

Figure 4.11: One can derive the single-step semantics with interrupts pending from the single-step atomic semantics for an ARM instruction

$$\frac{\begin{array}{l}\mathsf{INS\_SPEC\ LPC\_MODEL}\ p_1 * p_2\ \{ins\}\ q_1 * q_2 \\ \mathsf{SPEC\ LPC\_IRQ\_MODEL}\ iv_1 * iv_2\ isr\ iv_1 * iv_2 \\ q_2 \triangleright iv_1\end{array}}{\mathsf{SPEC\ LPC\_IRQ\_MODEL}\ p_1 * p_2 * iv_2\ (\{ins\} \cup isr)\ q_1 * iv_1 * iv_2}$$

Figure 4.12: Inference rule for integrating the invariant effect of an ISR

$$\frac{\begin{array}{l}\mathsf{INS\_SPEC\ LPC\_MODEL}\ p_1 * p_2\ \{ins\}\ q_1 * q_2 \\ \mathsf{SPEC\ LPC\_IRQ\_MODEL}\ ip_1 * iv_2\ isr\ iq_1 * iv_2 \\ q_2 \triangleright ip_1 \wedge iq_1 \triangleright iv_1 \wedge q_2 \triangleright iv_1\end{array}}{\mathsf{SPEC\ LPC\_IRQ\_MODEL}\ p_1 * p_2 * iv_2\ (\{ins\} \cup isr)\ q_1 * iv_1 * iv_2}$$

Figure 4.13: Inference rule for integrating the effect of an ISR

$$\frac{\begin{array}{l}\mathsf{SPEC}\ m\ p_1 * (f\, s_1)\ c\ q_1 * (f\, s_2) \\ p = \lambda\ x.f\ x * \mathsf{cond}\ (g\ x\ ) \\ g s_1 \Rightarrow g s_2\end{array}}{\mathsf{SPEC}\ m\ p_1 * (p\ \ s_1)\ c\ q_1 * (p\ \ s_2)}$$

Figure 4.14: Inference rule for introduing high-level assertion formula

CHAPTER 5

CORRECTNESS PROOF FOR THE SERIAL PORT ISR

An ISR for a hardware interrupt is a special kind of program in the context of verification. On one hand, as a standalone program, an ISR has its own specification to be verified. On the other hand, an ISR is more than a standalone program. Most of the time it is part of a device driver. Its cooperation with the main programs in the driver needs to be verified too.

An ISR usually involves device access. This increases the complexity, as the device state needs to be reasoned about. Usually, there are timing constraints on the behavior of an ISR. The most fundamental one is termination under some conditions. It is expected that an ISR always returns.

In this chapter, I present the correctness proof for the serial port ISR as a standalone program. The result will be used in the correctness proof of the serial port driver in Chapter 6. First, I introduce the extension to the serial port model in Section 3.2. This extension is to supports interrupts. Next, I explain the control flow of the serial port ISR. Then I introduce the formalization of the circular buffer used in the serial port driver. Finally I explain the correctness specification for the serial ISR and how the proof was achieved.

5.1 Serial Port Model with Interrupts

The serial port model in Section 3.2 is extended to support interrupts. Two registers: IER (interrupt enable register) and IIR (interrupt identifier register) are added, as shown in Table 5.1.

The UART0 in LPC2129 supports four interrupts. My model supported three of them. The one missing is due to that RBR only has a buffer size of 1 in my model. The wellformness is also extended to reflect that when an interrupt is requested, the corresponding status bit is also set:

$$\begin{aligned}
&\mathsf{uart\_wellform}\ s \triangleq \\
&(\neg s.\mathsf{LSR}_{\mathrm{TEMT}} \vee s.\mathsf{LSR}_{\mathrm{THRE}}) \wedge (s.\mathsf{clock} \neq 0 \vee s.\mathsf{LSR}_{\mathrm{THRE}}) \wedge \\
&s.\mathsf{clock} < \mathsf{get\_divisor}\ s \wedge (s.\mathsf{IIR}_{\mathrm{RDA}} \Rightarrow s.\mathsf{LSR}_{\mathrm{RDR}}) \wedge \\
&(s.\mathsf{IIR}_{\mathrm{THRE}} \Rightarrow s.\mathsf{LSR}_{\mathrm{THRE}}) \wedge (s.\mathsf{IIR}_{\mathrm{RLS}} \Rightarrow s.\mathsf{LSR}_{\mathrm{oe}}).
\end{aligned}$$

The interrupts in the serial port has different priorities. The details are shown in Table 5.2. An interrupt source can be enabled by setting the corresponding bit in IER. This command can occur at any moment of the serial port cycle. If it is at the beginning of the serial port cycle, the effect is immediate. That means firing the corresponding interrupt in this cycle if the corresponding bit in LSR is already set. Otherwise, the effect is delayed until the beginning of the next serial port cycle. When a bit in IER is cleared, the pending interrupt is not cleared.

5.1.1 Serial Port Interrupts Handling

The theorems in Figure 5.1 show that the model handles the interrupt source correctly. uart_has_irq x ps means interrupt x is raised. There may be more than one interrupt raised by the serial port at the same time. Only the one with the highest priority is visible to the processor.

(i) states that reading from RBR clears interrupt $\mathsf{IRQ}_{\mathrm{RDA}}$. (ii) states that reading from LSR clears interrupt $\mathsf{IRQ}_{\mathrm{RLS}}$. (iii) states that writing into from THR clears interrupt $\mathsf{IRQ}_{\mathrm{THRE}}$. (iv) states that, reading from IIR when interrupt $\mathsf{IRQ}_{\mathrm{THRE}}$ is the interrupt source, i.e., there are no other interrupts raised by the serial port, clears $\mathsf{IRQ}_{\mathrm{THRE}}$.

5.2 Serial Port ISR

The fundamental technique used in the proof is sequential composition. The first step is to find all the basic blocks. The first instruction of a basic block is its only entry point. Branches may only appear after the last instruction. Conditional execution does not count as branches.

The ISR is shown in Figure 5.2, and its control flow graph (CFG) is shown in

Figure 5.3. Arcs are marked with the conditions. Each basic block is named using the address of its first instruction.

Block 13c is the entry block. Here, registers R_1 through R_{12}, SPSR, and LR-4 are saved on the stack. Addresses of global variables are loaded into the registers. These variables include the parameters of the buffers as well as the tx flag.

Block 244 is the exit block; it restores the context. As a result, the processor mode is restored to what was at the point of this interrupt request except PC. PC is old PC minus 4.

From block 13c the control flow jumps to block 238. A loop forms between blocks 238 and 234. I call it the IRQ loop. The loop condition is that there are interrupts pending from the serial port. It is checked in block 238. If the condition is not satisfied, the loop breaks, and the control flow falls into the exit block 244. Otherwise, the control flow jumps to block 184 to do the work.

The IRQ loop body between blocks 184 and block 234 is a switch table with four entries. One for each of the three interrupt sources: IIR_{RLS}, IIR_{RDA}, and IIR_{THRE}, and one to deal with other situations. In block 184 the interrupt sources are read from register IIR. Depending on the values, the control flow dynamically jumps to different entries of the switch table.

IIR_{RLS} has the highest priority. It is handled in block 1c4. The interrupt is cleared by reading register LSR. The control flow then falls through block 230 and 234 to exit the switch table.

IIR_{RDA} has the second highest priority. It is handled in a loop between block 1d0 and 1cc. I call it the RDA loop. The loop condition is that the LSR_{RDR} bit is set in register LSR. This indicates whether there are any new characters at register RBR. In block 1d0, a new character in register RBR is copied into the slot following the *back* position of the rx buffer, and the *back* is increased by 1 if the buffer is not already full. At the end of this block register LSR is read to check the loop condition. If it is satisfied, the control flow jumps to block 1cc, where the *back* of the rx buffer is increased by 1. It then jumps to block 1d0 and the RDA loop starts over. If not, the loop breaks, and the control flow falls through and jumps to block 234 to exit the switch table.

IIR$_{\text{THRE}}$ has the lowest priority and is handled using a loop between block 218 and 208. I call it the THRE loop. The loop condition is that the LSR$_{\text{THRE}}$ bit is set in register LSR, which indicates that register THR is empty. It is checked in block 218. If the condition is satisfied, the control flow jumps to block 1f8 to do the work. Otherwise, it falls through block 224, block 230 and 234 to break the THRE loop and exit the switch table.

In block 1f8, whether the tx buffer is empty is checked . If the buffer is empty, the control flow jumps to block 234 to break the THRE loop and exit the switch table. Otherwise, the control flow falls through to block 208, where the first character in the tx buffer is copied to register THR, and the *front* of the buffer is updated. At this step the sending of this character is considered done. The control flow jumps back to block 218 and the THRE loop starts over again.

Blocks 194, 1f4 and 224 each have only one jump instruction. Block 234 is shared in all the cases. For the RDA loop, it updates the *back* of the rx buffer. For other cases, it reverts the effect of block 230 or some other instruction in block 1f4. The trick is that blocks 230 and 234 both have only one register move instruction and each reverts the other's effect.

5.3 Circular Buffer Model

A circular buffer is commonly used as a streaming buffer. It implements a queue on top of an array. There is more than one implementation. In this section I present a formalization of circular buffer used in the serial port driver at the assembly level.

5.3.1 Circular Buffer

The rx and tx buffers in the serial port driver are circular buffers. Figure 5.4 shows what such a circular buffer looks like.

In this implementation, it has four parameters:

1. *start*, which is the start address of the array.
2. *size*, which is the size of the array.
3. *front*, which is the array index to the position to dequeue.
4. *back*, which is the array index to the position to enqueue.

Front and *back* are relative to start.

There are four operations for a circular buffer:

1. Dequeue, which is to read the element from the *front* of the buffer and increase *front* by 1 modulo *size*.
2. Enqueue, which is to write an element to the *back* of the buffer and increase *back* by 1 modulo *size*.
3. Check whether the buffer is empty. In this implementation, a circular buffer is empty if and only if its *front* and *back* are equal.
4. Check whether the buffer is full. They are increased by 1 modulo *size* in enqueue and dequeue. In this implementation, a circular buffer is full if *back* plus 1 modulo *size* equals to *front*. This implies that the capacity of the buffer is 1 less than *size*, and *size* should be larger than 1 for the buffer to have any storage room.

5.3.2 Formalization of Circular Buffer

I formalized the circular buffer in Section 5.3.1 as a predicate over a memory region and some parameters in HOL. The memory address and array index are all in 32-bit words. The slot size is 1 byte. It is required that the size of the array is larger than 1. The definition is shown in Figure 5.5. Because parameters *start*, *size*, *front*, and *back* can be in memory or in register, they are specified as values, not as address-value pairs in the memory. The underlining array resides in domain *df*. *f* maps a byte value to an address. circBuf abstracts the buffer as a list *str*. The abstraction is recursively defined using dequeue.

The available space in the circular buffer is:

$$\begin{aligned}&\mathsf{circ\_space}\ \mathit{size}\ \mathit{front}\ \mathit{back} \triangleq\\ &\mathsf{if}\ \mathit{front} \leq_+ \mathit{back}\ \mathsf{then}\ \mathit{size} - (\mathit{back} - \mathit{front}) - 1\ \mathsf{else}\ \mathit{front} - \mathit{back} - 1).\end{aligned}$$

I proved the theorems in Figure 5.6 to describe the operations on the circular buffer at the ARM assembly level. They will be useful in automation later on. (i) maps the position in the list to the array index. (ii) states that the length of the list can be calculated from size, *front*, and *back* parameters. (iii) states that the buffer is empty when *front* = *back*. (iv) states that the length of the list is bounded by the capacity of the buffer. (v) states that the buffer is full when $(back +_{size} 1) = front$. (vi) states that dequeue takes the head off the list. (vii) states that enqueue appends a character at the tail of the list. Enqueue is not an atomic operation at machine level. Theorems (viii) and (ix) describe the intermediate steps in enqueue. (viii) states that, when a character is written at the *back* position of the queue, but *back* itself is not updated, the list remains the same. (ix) states that, when *back* is increased by 1 modulo *size*, the byte at the original *back* position is appended to the list.

5.4 Correctness of the Serial Port ISR

I use a high-level formula to assert the behavior of the ISR. By high-level I mean using the data structures usually seen in programs in high-level programming languages. In Chapter 3 the polling-based driver was asserted in a similar way. However, back then the high-level assertion formula was mainly used to describe the serial port state. Here, I will reuse the formulas in Chapter 3, and augment them with the circular buffer models to address the high-level data structure in the driver itself.

5.4.1 High-Level View of the ISR Behavior

I use a high-level construct aU to specify the serial port ISR behavior. It groups together the involved resources: the serial port state, the rx buffer, and the tx buffer, as well as the constraint on these resources. Doing this helps automation, because parsing of the theorems is easier now. It also helps readability.

aU is defined in Figure 5.7. Different parts are individually grouped and labeled for convenience. Of those (i) defines the resources involved: memory regions for the buffers and the serial port state. All other parts are constraints on the resources. In (ii), some elements of the device state are lifted out to make weakening process easier. The wellformness requirement for the device state is defined in (iii). Timing property is defined in (iv). The properties of tx and rx are defined in (v) and (vi) .

5.4.2 Timing Property

Termination needs be proved for any ISR. In this case, I need to prove that the IRQ loop terminates. In turn, the termination of the THRE loop and the RDA loop needs be proved.

Intuitively, if the serial port receives data at too fast a rate, the RDA loop and IRQ loop may not terminate (the termination of the THRE loop does not depend on the device speed because the tx buffer will be emptied and $\mathsf{IIR}_{\text{THRE}}$ will not be set again unless THR is filled again), because new interrupts will always appear in IIR during the loop. The termination could be proved under the condition that the device is so slow that the IRQ loop can be finished inside 1 clock cycle of the serial port.

The brute force approach, like what I did before, can be used here. However, due to the complex code structure, it will take too much effort. Every execution point needs to be examined because the autonomous transition may occur at any moment.

The alternate approach is to introduce a strong assumption about the timing when an interrupt becomes pending. It is assumed here that no autonomous transition occurs inside the execution of the ISR. At first this assumption may appear as too strong, but it is quite reasonable. Put another way, this assumption bounds the latency in handling the interrupt requests. (iv) means there are at least $\mathit{cushion}_{\min} + \mathit{cushion}$ CPU cycles before the next autonomous transition.

Even though *cushion* and $\mathit{cushion}_{\min}$ have the same mathematical meaning, in practice, they serve different functions. $\mathit{cushion}_{\min}$ is used to indicate the minimum cushion of time before the next autonomous transition, and appears as a constant. *cushion* is a free variable which can be instantiated in sequential composition and branch combination. Constant offset can be added to *cushion* in both precondition and postcondition based on the following theorems in Figure 5.8.

I use aU to assert the precondition and postcondition of program segments of the ISR. *cushion* in the precondition is always larger than the one in the postcondition. The difference indicates the upper bound of the execution path of the program. For the whole ISR, this difference is equal to the longest path[1] of the ISR.

The timing constraint has monotonicity, as shown in Figure 5.9. The monotonicity is helpful in sequential composition and branch combination. In sequential composition, it can be used to weaken the postcondition or strengthen the precondition of

[1]It is actually worst-case execution time(WCET).

the Hoare triples. If, for different branches, the path lengths are not equal, the larger value is chosen for the assertion of the combined program.

In addition to making the proof manageable, a timing constraint is needed to prove the ISR behavior in which no incoming data is dropped due to the latency in the serial port interrupt handling, because no new data arrives in the execution of the ISR.

5.4.3 tx Property

In (v) the tx state is specified using str_{txout} and str_{txbuf}. str_{txout} is the string abstraction of register THR (when $\mathsf{LSR}_{\mathrm{THRE}}$ is not set) and the outgoing stream in the serial port device, as formalized using sentString, while str_{txbuf} is the string in the tx buffer, as formalized using circBuf. Safety in tx means that the string sent out must come from the tx buffer. It is guaranteed by str_{tx} being an invariant. So no additional assertion is necessary.

5.4.4 rx Property

rx property is described at (vi). It has two parts. One is the abstraction of strings from the rx pipeline and buffer. The other is the safety property of the rx function.

5.4.4.1 String Abstraction

Two strings are abstracted from the rx part of the system state. One is str_{rxin}, which is the string from RBR (if $\mathsf{LSR}_{\mathrm{RDR}}$ is set) and the incoming stream. This is formalized in the inputString predicate at (vi). The other is str_{rxbuf}, which is the string in the rx buffer. This is formalized in the circBuf predicate at (vi).

If no incoming characters are dropped, str_{rx} should be an invariant. If $\mathsf{LSR}_{\mathrm{RDR}}$ is set, str_{rxin} includes the character in RBR, so it must not be NULL.

5.4.4.2 rx Safety

rx safety means that no characters are inserted to the rx buffer from outside the income stream. To formulate this notion in a Hoare triple, I define a predicate which asserts that the string in the rx buffer comes from the rx buffer and the device state at some previous moment. The rx buffer safety is established for a program segment when the rx buffers in the precondition and postcondition come from the rx buffer and the device state at the same previous moment.

Assume there is a reference time point[2] t_0 when $\mathsf{LSR}_{\mathrm{RDR}}$ is not set. At any moment after t_0, str_{rxbuf} can be split into two parts so that the first part equals to str_{rxbuf0}, the string in the buffer at t_0. The second part and the character in RBR (if $\mathsf{LSR}_{\mathrm{RDR}}$ is set) should be sampled from the income stream $strm_{\mathrm{rx0}}$ at t_0. This sampling notion is defined in sample in Figure 5.10. Here, str_{buf} is the second part of a string stored in a buffer. Possible dropping of characters are accounted for by allowing n to be larger than 0 in the recursion. The rx safety property is defined in ishifted in Figure 5.10.

5.4.5 Hoare Triple for the Serial Port ISR

The Hoare triple for the ISR is shown below:

$$\vdash \mathsf{SPEC\ LPC\_IRQ\_MODEL}$$

$$(\mathsf{aPC}\ 316 * \mathsf{aPSR\ ips}\ ipsr * \mathsf{aR\ LRirq}\ lrirq * \tag{5.4i}$$

$$\left.\begin{array}{l}\mathsf{aMD}\ 18 * \mathsf{aCPSR}\ cpsr * \mathsf{aS1\ PSR_A}\ psra * \mathsf{aS1\ PSR_C} psrc * \\ \mathsf{aS1\ PSR_I}\ T * \mathsf{aS1\ PSR_N}\ psrn * \\ \mathsf{aS1\ PSR_Q}\ psrq * \mathsf{aS1\ PSR_V}\ psrv * \mathsf{aS1\ PSR_Z}\ psrz *\end{array}\right\} \tag{5.4ii}$$

$$\left.\begin{array}{l}\mathsf{aPSTACK\ irq}\ spirq\ [] \\ [w_1; w_3; w_5; w_7; w_9; w_{11}; w_{13}; w_{15}; w_{17}; w_{19}; w_{21}; w_{23}; w_{25}; w_{27}; stk] *\end{array}\right\} \tag{5.4iii}$$

$$\left.\begin{array}{l}\mathsf{aR\ R_0} r_0 * \mathsf{aR\ R_1}\ r_1 * \mathsf{aR\ R_2}\ r_2 * \mathsf{aR\ \_3}\ r_3 * \mathsf{aR\ R_4}\ r_4 * \\ \mathsf{aR\ R_5}\ r_5 * \mathsf{aR\ R_6}\ r_6 * \mathsf{aR\ R_7}\ r_7 * \mathsf{aR\ R_8}\ r_8 * \\ \mathsf{aR\ R_9}\ r_9 * \mathsf{aR\ R_{10}}\ r_{10} * \mathsf{aR\ R_{11}}\ r_{11} * \mathsf{aR\ R_{12}}\ r_{12} *\end{array}\right\} \tag{5.4iv}$$

$$\mathsf{aBYTE\_MEMORY}\ dff * \tag{5.4v}$$

$$\left.\begin{array}{l}\mathsf{aU}\ divisor\ 85\ cushion_{\mathrm{min}}\ strm_{\mathrm{rx0}}\ str_{\mathrm{rxbuf0}}\ str_{\mathrm{tx}}\ str_{\mathrm{txout}}\ str_{\mathrm{txbuf}}\ str_{\mathrm{rx}} \\ str_{\mathrm{rxin}}\ str_{\mathrm{rxbuf}}\ \underline{start_{\mathrm{rx}}}\ \underline{size_{\mathrm{rx}}}\ \underline{start_{\mathrm{tx}}}\ \underline{size_{\mathrm{tx}}}\ thre\ ie_{\mathrm{rls}}\ ie_{\mathrm{rda}}\ ie_{\mathrm{thre}}\ irqs \\ df_{\mathrm{rx}}\ f_{\mathrm{rx}}\ df_{\mathrm{tx}}\ f_{\mathrm{tx}}\ \underline{front_{\mathrm{rx}}}\ \underline{back_{\mathrm{rx}}}\ \underline{front_{\mathrm{tx}}}\ \underline{back_{\mathrm{tx}}}\ ps *\end{array}\right\} \tag{5.4vi}$$

$$\mathsf{cond}(ipsr.\mathsf{M} \in \{16, 17, 18, 19, 23, 27, 31\} \wedge \tag{5.4vii}$$

[2]It always exists.

$$\mathit{ipsr}.\mathsf{F} \wedge (\mathit{ipsr}.\mathsf{IT} = 0) \wedge \neg \mathit{ipsr}.\mathsf{E} \wedge \neg \mathit{ipsr}.J \wedge \neg \mathit{ipsr}.\mathsf{T} \wedge \tag{5.4viii}$$

$$\left.\begin{array}{l} (\overleftarrow{\underline{\mathit{front\_addr}_{\mathrm{rx}}}}^{2} \cup \overleftarrow{\underline{\mathit{back\_addr}_{\mathrm{rx}}}}^{2} \cup \overleftarrow{\underline{\mathit{front\_addr}_{\mathrm{tx}}}}^{2} \cup \\ \overleftarrow{\underline{\mathit{back\_addr}_{\mathrm{tx}}}}^{2} \cup \overleftarrow{\underline{\mathit{tx\_running\_addr}}}^{4}) \subseteq \mathit{df} \wedge \end{array}\right\} \tag{5.4ix}$$

$$\left.\begin{array}{l} (3 \,\&\&\, \mathit{lrirq} - 4 = 0) \wedge \\ (3 \,\&\&\, \mathit{spirq} - 60 = 0) \wedge \\ (3 \,\&\&\, \mathit{spirq} - 56 = 0))) \end{array}\right\} \tag{5.4x}$$

$$\mathsf{isr\_code} \tag{5.4xi}$$

$$\left.\begin{array}{l} (\mathsf{aPC}\ (\mathit{lrirq} - 4) * \mathsf{aPSR}\ \mathit{ips}\ \mathit{ipsr} * \ * \\ \mathsf{aR}\ \mathsf{LRirq}\ (\mathit{lrirq} - 4) * \end{array}\right\} \tag{5.4xii}$$

$$\left.\begin{array}{l} \mathsf{aMD}\ \mathit{ipsr}.\mathsf{M} * \mathsf{aCPSR}\ (\mathit{cpsr}\ \mathit{with}\ \mathsf{ge} := \mathit{ipsr}.\mathsf{GE}) * \mathsf{aS1}\ \mathsf{PSR}_{\mathrm{A}}\ \mathit{ipsr}.\mathsf{A} * \\ \mathsf{aS1}\ \mathsf{PSR}_{\mathrm{C}}\ \mathit{ipsr}.\mathsf{C} * \mathsf{aS1}\ \mathsf{PSR}_{\mathrm{I}}\ \mathit{ipsr}.\mathsf{I} * \mathsf{aS1}\ \mathsf{PSR}_{\mathrm{N}}\ \mathit{ipsr}.\mathsf{N} * \\ \mathsf{aS1}\ \mathsf{PSR}_{\mathrm{Q}}\ \mathit{ipsr}.\mathsf{Q} * \mathsf{aS1}\ \mathsf{PSR}_{\mathrm{V}}\ \mathit{ipsr}.\mathsf{V} * \mathsf{aS1}\ \mathsf{PSR}_{\mathrm{Z}}\ \mathit{ipsr}.\mathsf{Z} * \end{array}\right\} \tag{5.4xiii}$$

$$\left.\begin{array}{l} \mathsf{aPSTACK}\ \mathsf{irq}\ \mathit{spirq}\ [] \\ [\mathit{lrirq} - 4; r_{12}; r_{11}; r_{10}; r_9; r_8; r_7; r_6; r_5; r_4; r_3; \\ r_2; r_1; r_0; \mathrm{encode\_psr}\ \mathit{ipsr}] \end{array}\right\} \tag{5.4xiv}$$

$$\left.\begin{array}{l} \mathsf{aR}\ \mathsf{R}_0\ r_0 * \mathsf{aR}\ \mathsf{R}_1\ r_1 * \mathsf{aR}\ \mathsf{R}_2\ r_2 * \mathsf{aR}\ \_3\ r_3 * \mathsf{aR}\ \mathsf{R}_4\ r_4 * \\ \mathsf{aR}\ \mathsf{R}_5\ r_5 * \mathsf{aR}\ \mathsf{R}_6\ r_6 * \mathsf{aR}\ \mathsf{R}_7\ r_7 * \mathsf{aR}\ \mathsf{R}_8\ r_8 * \\ \mathsf{aR}\ \mathsf{R}_9\ r_9 * \mathsf{aR}\ \mathsf{R}_{10}\ r_{10} * \mathsf{aR}\ \mathsf{R}_{11}\ r_{11} * \mathsf{aR}\ \mathsf{R}_{12}\ r_{12} * \end{array}\right\} \tag{5.4xv}$$

$$\mathsf{aBYTE\_MEMORY}\ \mathit{df}\ \underline{\mathit{newf}} * \tag{5.4xvi}$$

$$\neg \mathsf{aU}\ \mathit{divisor}\ 0\ \mathit{cushion}_{\mathrm{min}}\ \mathit{strm}_{\mathrm{rx0}}\ \mathit{str}_{\mathrm{rxbuf0}}\ \mathit{str}_{\mathrm{tx}} \tag{5.4xvii}$$

$$\left.\begin{array}{l} (\mathsf{if\ MEM\ IIR}_{\mathrm{THRE}}\ \mathit{irqs} \\ \quad \mathsf{then\ if}\ \underline{\mathit{back}_{\mathrm{tx}}} = \underline{\mathit{front}_{\mathrm{tx}}}\ \mathsf{then}\ \mathit{str}_{\mathrm{txout}}\ \mathsf{else\ HD}\ \mathit{str}_{\mathrm{txbuf}} :: \mathit{str}_{\mathrm{txout}} \\ \quad \mathsf{else}\ \mathit{str}_{\mathrm{txout}}) \end{array}\right\} \tag{5.4xviii}$$

$$\left.\begin{array}{l} (\mathsf{if\ MEM\ IIR}_{\mathrm{THRE}}\ \mathit{irqs} \\ \quad \mathsf{then\ if}\ \underline{\mathit{back}_{\mathrm{tx}}} = \underline{\mathit{front}_{\mathrm{tx}}}\ \mathsf{then}\ \mathit{str}_{\mathrm{txbuf}}\ \mathsf{else}\ TL\ \mathit{str}_{\mathrm{txbuf}} \\ \quad \mathsf{else}\ \mathit{str}_{\mathrm{txbuf}}) \end{array}\right\} \tag{5.4xix}$$

$$\left.\begin{array}{l}(\textsf{if MEM IIR}_{\mathrm{RDA}}\ \mathit{irqs}\\ \quad\textsf{then if } (63\ \&\&\ \underline{\mathit{back}_{\mathrm{rx}}} + 1) \neq \underline{\mathit{front}_{\mathrm{rx}}}\\ \qquad\qquad\textsf{then } \mathit{str}_{\mathrm{rx}}\\ \qquad\qquad\textsf{else } \mathit{str}_{\mathrm{rxbuf}}\textsf{++}TL\ \mathit{str}_{\mathrm{rxin}}\\ \quad\textsf{else } \mathit{str}_{\mathrm{rx}})\end{array}\right\} \tag{5.4xx}$$

$$(\textsf{if MEM IIR}_{\mathrm{RDA}}\ \mathit{irqs}\ \textsf{then TL}\ \mathit{str}_{\mathrm{rxin}}\ \textsf{else}\ \mathit{str}_{\mathrm{rxin}}) \tag{5.4xxi}$$

$$\left.\begin{array}{l}(\textsf{if MEM IIR}_{\mathrm{RDA}}\ \mathit{irqs}\\ \quad\textsf{then if } (63\ \&\&\ \underline{\mathit{back}_{\mathrm{rx}}} + 1) \neq \underline{\mathit{front}_{\mathrm{rx}}}\\ \qquad\qquad\textsf{then } \mathit{str}_{\mathrm{rxbuf}}\textsf{++}[\textsf{HD}\ \mathit{str}_{\mathrm{rxin}}]\\ \qquad\qquad\textsf{else } \mathit{str}_{\mathrm{rxbuf}}\\ \quad\textsf{else } \mathit{str}_{\mathrm{rxbuf}})\end{array}\right\} \tag{5.4xxii}$$

$$\underline{\mathit{start}_{\mathrm{rx}}}\ \underline{\mathit{size}_{\mathrm{rx}}}\ \underline{\mathit{start}_{\mathrm{tx}}}\ \underline{\mathit{size}_{\mathrm{tx}}} \tag{5.4xxiii}$$

$$(\textsf{if MEM IIR}_{\mathrm{THRE}}\ \mathit{irqs}\ \textsf{then}\ \underline{\mathit{back}_{\mathrm{tx}}} = \underline{\mathit{front}_{\mathrm{tx}}}\ \textsf{else}\ \mathit{thre}) \tag{5.4xxiv}$$

$$\mathit{ie}_{\mathrm{rls}}\ \mathit{ie}_{\mathrm{rda}}\ \mathit{ie}_{\mathrm{thre}}\ []\ \mathit{df}_{\mathrm{rx}} \tag{5.4xxv}$$

$$\left.\begin{array}{l}(\textsf{if MEM IIR}_{\mathrm{RDA}}\ \mathit{irqs}\\ \quad\textsf{then } (\underline{\mathit{back}_{\mathrm{rx}}} + \underline{\mathit{start}_{\mathrm{rx}}} \mapsto \textsf{HD}\ \mathit{str}_{\mathrm{rxin}})\ f_{\mathrm{rx}}\\ \quad\textsf{else } f_{\mathrm{rx}})\end{array}\right\} \tag{5.4xxvi}$$

$$\mathit{df}_{\mathrm{tx}}\ f_{\mathrm{tx}}\ \underline{\mathit{front}_{\mathrm{rx}}} \tag{5.4xxvii}$$

$$\left.\begin{array}{l}(\textsf{if MEM IIR}_{\mathrm{RDA}}\ \mathit{irqs}\\ \quad\textsf{then if } (63\ \&\&\ \underline{\mathit{back}_{\mathrm{rx}}} + 1) \neq \underline{\mathit{front}_{\mathrm{rx}}}\\ \qquad\qquad\textsf{then } 63\ \&\&\ \underline{\mathit{back}_{\mathrm{rx}}} + 1\\ \qquad\qquad\textsf{else } \underline{\mathit{back}_{\mathrm{rx}}}\\ \quad\textsf{else } \underline{\mathit{back}_{\mathrm{rx}}})\end{array}\right\} \tag{5.4xxviii}$$

$$\left.\begin{array}{l}(\textsf{if MEM IIR}_{\mathrm{THRE}}\ \mathit{irqs}\\ \quad\textsf{then if } \underline{\mathit{back}_{\mathrm{tx}}} = \underline{\mathit{front}_{\mathrm{tx}}}\ \textsf{then}\ \underline{\mathit{front}_{\mathrm{tx}}}\ \textsf{else}\ 127\ \&\&\ \underline{\mathit{front}_{\mathrm{tx}}} + 1\\ \quad\textsf{else } \underline{\mathit{front}_{\mathrm{tx}}})\end{array}\right\} \tag{5.4xxix}$$

$$\underline{\mathit{back}_{\mathrm{tx}}}). \tag{5.4xxx}$$

The placeholders used in Equation 5.4 is listed in Table 5.3.

When ARM processor takes an interrupt request, the hardware will perform some preparation. The effects are shown in the precondition of the Hoare triple for the ISR. At (5.4i) PC, and PSR and LR in IRQ mode are specified. Fields in CPSR are specified at (5.4ii).

Once inside the ISR, registers to be used are saved onto the stack. Before the push, the stack is specified at (5.4iii). It is assumed that enough space is on the stack in IRQ mode. The nonbanked registers are specified at (5.4iv).

Memory region (*df*,*f*) is specified at (5.4v). It is for the storage of the parameters of the rx and tx buffers and the tx_running flag.

The state of the serial port device and the rx and tx buffers are specified at (5.4vi). *cushion* is specified as 85 here, because ISR finishes its execution in 85 cycles along the longest path. This guarantees that no new interrupts will be raised during the execution of the ISR.

There are constraints on the preconditions specified in the cond term. First, the value in PSR of IRQ mode must be valid for CPSR. (5.4vii) constrains the possible value for the mode field, and (5.4viii) constrains some other fields. Since a copy of previous CPSR is stored in PSR of IRQ mode, this condition is satisfied. Then memory domain *df* is constrained at (5.4ix). Finally, alignment constrains are specified at (5.4x).

The postcondition describes what the system state is after the execution of the ISR. There are several points:

1. The ISR terminates. As shown at (5.4xii), PC eventually points to the next instruction at the time when the interrupt request was taken.
2. The context switch is working properly. CPSR is restored to its original state as shown at (5.4xiii). This means the processor is in the original mode, and all the mask and status bits are restored as well. The data pushed to the stack are popped. The stack in IRQ mode is restored at (5.4xiv) and the unbanked registers are restored at (5.4xv). The memory safety of stack is assumed in precondition at (5.4iii).
3. The interrupts are properly handled.

(a) Interrupts are all cleared as shown at (5.4xxv).

(b) *Front* and *back* for rx and tx buffers and tx_running flag are updated. The exact low-level detail in terms of memory update is too lengthy to be shown here, due to a combination of different conditions: whether there are interrupts pending, which interrupt source is pending, whether the rx buffer is full, whether the tx buffer is empty, and whether tx_running flag is set. Byte-level memory configuration does not help either. I use *newf* as the placeholder for the exact term at (5.4xvi).

(c) For the rx function, updates on the strings str_{rx}, str_{rxin}, str_{rxbuf} are shown at (5.4xx), (5.4xxi) and (5.4xxii). The low-level detail in terms of the buffer memory and its *back* value are shown at (5.4xxvi) and (5.4xxviii).
Nothing happens unless interrupt $\mathsf{IIR}_{\mathrm{RDA}}$ is pending. Otherwise, there are two scenarios. No character is dropped when the character from RBR is copied to the rx buffer. This happens under these three conditions: the speed of the serial port is sufficiently slow, the interrupt handling is not delayed too much, and the rx buffer has space available. The first two conditions are assumed in the timing constraints in aU term at (5.4vi).
If the buffer is full, the character is copied to the *back* position of the array in the buffer without updating the value of *back*, so it is discarded. The rx buffer safety is guaranteed by the invariance shown in $strm_{\mathrm{rx0}}$ and str_{rxbuf0} at (5.4xvii).

(d) For the tx function, updates on the strings str_{tx}, str_{txout}, str_{txbuf} are shown at (5.4xvii), (5.4xviii) and (5.4xix). The low-level detail in terms of the *front* value of tx buffer is shown at (5.4xxix). Nothing happens unless interrupt $\mathsf{IIR}_{\mathrm{THRE}}$ is pending. Otherwise, if tx buffer is not empty, the first character from the tx buffer is sent out, and $\mathsf{LSR}_{\mathrm{THRE}}$ is cleared at (5.4xxiv). The character will never be lost, since str_{tx} is an invariant.

5.5 Proof

A significant part of the correctness proof for the ISR is done interactively, except for the basic semantics for basic blocks. In this section I discuss some technical issues related to the proof process, from the strategy at high-level to a specific example of a difficult program fragment, the IRQ loop.

5.5.1 Proof Approach

I proved the Hoare triple for the ISR (Equation 5.4) from bottom up following the control flow graph as shown in Figure 5.3:

1. At the ground level, Hoare triples for each instruction are proved automatically as described in last chapter.
2. Based on Hoare triples for each instruction, Hoare triples for the basic blocks (shown in Figure 5.3) are proved using sequential composition mostly automatically.
3. With Hoare triples for the basic blocks ready, Hoare triples for the THRE loop, the RDA loop and the RLS case in the switch table are proved manually using sequential composition and branch combination.
4. With theorems for these three cases, the Hoare triple for the IRQ loop is proved using induction manually.
5. Finally, based on Hoare triples for the entry block, the IRQ loop, and the exit block, the Hoare triple for the ISR is proved using sequential composition automatically.

5.5.1.1 High-Level Assertion Formula

aU is designed to specify the Hoare triple of the ISR as a whole. It is not suitable to be used to specify Hoare triples for individual instructions, because some invariants included in aU assertion may not hold at every instruction of the ISR. So, aU assertion can only be introduced for a code block where such invariants hold in the precondition and postcondition.

A high-level assertion formula such as aU congregates a group of resource specifications and constraints. It can be introduced to an existing Hoare triple following these steps:

1. For every resource specification or constraint in the assertion formula but not in the Hoare triple, it is introduced into both precondition and postcondition symmetrically using the frame rule.

2. Similarly, every missing constraint can be introduced into the cond term in the precondition of the Hoare triple using precondition strengthening.
3. A constraint can be introduced into the postcondition of the Hoare triple using the following theorem:

$$\begin{aligned}&\vdash (g_1 \;\Rightarrow\; g_2) \Rightarrow \\ &\quad \mathsf{SPEC}\; m\, (p \;* \mathsf{cond}\; g_1)\; c\; q \;\Rightarrow \mathsf{SPEC}\; m\, (p \;* \mathsf{cond}\; g_1)\; c\, (q \;* \mathsf{cond}\; g_2). \qquad (5.5)\end{aligned}$$

   Here, constraint g_2 can be introduced to the postcondition of a Hoare triple if it can be implied by the constraint g_1 from the precondition.
4. The high-level assertion formula can now be introduced using its definition.

5.5.1.2 Intermediate Assertion

Introducing a high-level assertion formula often requires manual proof of a lemma of the form $g_1 \;\Rightarrow\; g_2$, as in Equation 5.5. So it helps to have formula g_1 and g_2 be as small as possible. I designed several stripped-down versions of aU for different code blocks. They are intermediate assertions from which aU is to be introduced. One of them is aU_e:

$$\begin{aligned}&\mathsf{aU}_\mathrm{e}\; \mathit{divisor}\; \mathit{cushion}\; \mathit{cushion}_\mathrm{min}\; \mathit{strm}_\mathrm{rx0}\; \mathit{str}_\mathrm{rxbuf0}\; \mathit{thre}\; \mathit{ie}_\mathrm{rls}\; \mathit{ie}_\mathrm{rda}\; \mathit{ie}_\mathrm{thre}\; \mathit{irqs}\; \mathit{str}_\mathrm{rxin} \\ &\mathit{str}_\mathrm{rxbuf}\; \mathit{str}_\mathrm{txout}\; \mathit{ps} \triangleq \\ &\mathsf{aP}\; \mathit{ps} * \mathsf{cond}\; ((\mathsf{get\_divisor}\; \mathit{ps} = \mathit{divisor}) \wedge (\mathsf{uart\_pending\_irqs}\; \mathit{ps} = \mathit{irqs}) \wedge \\ &(\mathit{ps}.\mathsf{IER}_\mathrm{RLS} = \mathit{ie}_\mathrm{rls}) \wedge (\mathit{ps}.\mathsf{IER}_\mathrm{RDA} = \mathit{ie}_\mathrm{rda}) \wedge \\ &(\mathit{ps}.\mathsf{IER}_\mathrm{THRE} = \mathit{ie}_\mathrm{thre}) \wedge (\mathit{ps}.\mathsf{LSR}_\mathrm{THRE} = \mathit{thre}) \wedge \\ &\mathsf{uart0.wellform}\; \mathit{ps} \wedge \neg \mathit{ps}.\mathsf{LCR}_\mathrm{DLAB} \wedge \\ &\mathit{ps}.\mathsf{clock} + \mathit{cushion}_\mathrm{min} + \mathit{cushion} \; < \mathsf{get\_divisor}\; \mathit{ps} \wedge \\ &\mathsf{sentString}\; \mathit{str}_\mathrm{txout}\; \mathit{ps} \wedge \\ &\mathsf{inputString}\; \mathit{str}_\mathrm{rxin}\; \mathit{ps} \wedge \\ &(\mathit{ps}.\mathsf{LSR}_\mathrm{RDR} \Rightarrow \neg \mathsf{NULL}\; \mathit{str}_\mathrm{rxin}) \wedge \mathsf{ishifted}\; \mathit{strm}_\mathrm{rx0}\; \mathit{str}_\mathrm{rxbuf0}\; \mathit{ps}\; \mathit{str}_\mathrm{rxbuf})\end{aligned}$$

aU_e is intended to specify the entry block, the exit block, and the RLS case in the switch table. In these code blocks no memory besides the stack is accessed. Compared to aU, aU_e leaves out memory specification and constraints on the rx and tx buffers. It only keeps the constraints involving the serial port state *ps*. Those specifications and constraints left out can be easily added back using the frame rule. So it does not take any manual proof to introduce aU from aU_e. This technique reduces the effort in manual proof. Similar variants of aU are designed to be used in the proof of the RDA loop and THRE loop.

5.5.2 Proof of the IRQ Loop

The IRQ loop is a complex yet interesting program fragment. The loop condition depends on device register IIR; its termination depends on the speed of the serial port; it has complex control flow, especially the body of the loop is a switch table. In this section I discuss how I proved correctness for it.

5.5.2.1 Switch Table

The switch table has three cases: the RLS case, the RDA case and the THRE case. Each case is only executed if the corresponding interrupt source is at the head of the pending interrupt list. After the execution of one case, the corresponding interrupt source is popped from the pending interrupt list.

The RDA case and the THRE case are implemented as loops. But both loops only execute once, because in my serial port model the RBR buffer and THR buffer has a size of 1, and the timing constraint implies that no new interrupts are raised inside the ISR. So for all three cases, I used sequential composition to obtain the low-level Hoare triples, then introduced aU by introducing the simpler variants of aU first. There are two branches in the THRE case based on whether tx buffer is empty or not. An additional pass of branch combination was applied on it.

5.5.2.2 IRQ Loop

I manually combined the three cases in the switch table to produce the Hoare triple assertion for the IRQ loop. Specifically, in the assertion I associate *cushion* in aU formula with the pending interrupts list *irqs* using $\mathsf{latency}$, which is defined in Figure 5.11. In the definition, 24, 19, and 12 are path lengths for the THRE case, the RDA case and the RLS case respectively. $\mathsf{latency}$ keeps track of the longest path of the ISR based on the pending interrupt sources.

The Hoare triple is proved by induction on the pending interrupt list. Because of the timing constraint, no new interrupt requests occur inside the ISR. The head of the pending interrupt list will be popped in each loop. That is how the loop terminates.

Since the loop body is a switch table, the inductive step has to deal with its three cases. Each case has a different path length. In Hoare triples for these three cases, cushion equals the respective path length in the precondition, and zero in the postconditions. Using theorems in Figure 5.8, latency can be introduced to the Hoare triples for these three cases. These Hoare triples can now be used as lemmas in the inductive step.

latency is bounded:

$$\vdash \forall\, ps\ irqs.(\mathsf{uart\_pending\_irqs}\ ps = irqs) \Rightarrow \mathsf{latency}\ irqs \leq 55$$

It is proved based on the fact that all pending interrupts are distinct, thus the length of pending irq list is finite. Using this theorem and the monotonicity theorem in Figure 5.9, the latency term in the precondition of Hoare triple for the IRQ loop is replaced with 55.

Table 5.1: Serial port model with interrupts. R indicates read-only registers; W indicates write-only. RW indicates no restriction on access. The first four columns are from data in the LPC2129 manual.

Register	Address offset	Function	Access	When is read undefined?	When is write undefined?	Side-effect of read	Side-effect of write
RBR	0	Receiver buffer when $\neg\mathsf{LCR}_{\mathrm{DLAB}}$	R	No data received	Never	Reset $\mathsf{LSR}_{\mathrm{RDR}}$	None
THR	0	Transmit holding when $\neg\mathsf{LCR}_{\mathrm{DLAB}}$	W	Never	No room for transmission	None	Reset $\mathsf{LSR}_{\mathrm{THRE}}$
DLL	0	Divisor latch LSB when $\mathsf{LCR}_{\mathrm{DLAB}}$	RW	Never	Never	None	None
IER	0	Interrupt enable when $\neg\mathsf{LCR}_{\mathrm{DLAB}}$	RW	Never	Write ones to bits 7:3	None	Disable or enable the interrupts
DLM	4	Divisor latch MSB when $\mathsf{LCR}_{\mathrm{DLAB}}$	RW	never	Never	None	None
IIR	8	Interrupt ID	R	Never	Always	Clear $\mathsf{IIR}_{\mathrm{THRE}}$	None
FCR	8	FIFO control	W	Always	Overwrite reserved bits or disable FIFOs	None	Reset transmission or receiving queue and flags
LCR	12	Line control	RW	Never	Never	None	Assign $\mathsf{LCR}_{\mathrm{DLAB}}$ flag
LSR	20	Line status	R	Never	Always	Reset $\mathsf{LSR}_{\mathrm{OE}}$	None
SCR	28	Scratch pad	RW	Never	Never	None	None

Table 5.2: Serial port model interrupts coverage. The first two columns are adapted from the LPC2129 manual.

Interrupt	Register	Priority	Interrupt Source	Interrupt Set	Interrupt Reset
$\mathsf{IRQ}_{\mathrm{RLS}}$	$\mathsf{IIR}_{\mathrm{RLS}}$	Highest	$\mathsf{LSR}_{\mathrm{oe}}$	New data is received when $\mathsf{LSR}_{\mathrm{RDR}}$ and $\mathsf{IER}_{\mathrm{RLS}}$	LSR is read
$\mathsf{IRQ}_{\mathrm{RDA}}$	$\mathsf{IIR}_{\mathrm{RDA}}$	Second	$\mathsf{LSR}_{\mathrm{RDR}}$	rx data in RBR when $\mathsf{IER}_{\mathrm{RDA}}$	Read RBR or reset the rx
$\mathsf{IRQ}_{\mathrm{THRE}}$	$\mathsf{IIR}_{\mathrm{THRE}}$	Third	$\mathsf{LSR}_{\mathrm{THRE}}$	tx data sent from THR when $\neg\mathsf{LSR}_{\mathrm{THRE}}$ and $\mathsf{IER}_{\mathrm{THRE}}$	Read IIR when $\mathsf{IIR}_{\mathrm{THRE}}$ is the current interrupt source, or write THR, or reset tx

$$\vdash \neg\mathsf{uart\_dlab}\ ps\ \wedge\ (addr = \mathsf{0xE000C000}) \wedge \mathsf{uart\_has\_irq}\ \mathsf{IRQ}_{\mathrm{RDA}}\ ps \Rightarrow$$
$$\neg\mathsf{uart\_has\_irq}\ \mathsf{IRQ}_{\mathrm{RDA}}\ (\mathsf{SND}\ (\mathsf{SND}\ (\mathsf{uart0.mapped\_read}\ addr\ ps))) \qquad \text{(i)}$$
$$\vdash (addr = \mathsf{0xE000C014}) \wedge \mathsf{uart\_has\_irq}\ \mathsf{IRQ}_{\mathrm{RLS}}\ ps \Rightarrow$$
$$\neg\mathsf{uart\_has\_irq}\ \mathsf{IRQ}_{\mathrm{RLS}}\ (\mathsf{SND}\ (\mathsf{SND}\ (\mathsf{uart0.mapped\_read}\ addr\ ps))) \qquad \text{(ii)}$$
$$\vdash \neg\mathsf{uart\_dlab}\ ps \wedge (addr = \mathsf{0xE000C000}) \wedge \mathsf{uart\_has\_irq}\ \mathsf{IRQ}_{\mathrm{THRE}}\ ps \Rightarrow$$
$$\neg\mathsf{uart\_has\_irq}\ \mathsf{IRQ}_{\mathrm{THRE}}\ (\mathsf{SND}\ (\mathsf{uart0.mapped\_write}\ addr\ [d]\ ps)) \qquad \text{(iii)}$$
$$\vdash \neg\mathsf{uart\_dlab}\ ps\ \wedge (addr = \mathsf{0xE000C008}) \wedge \mathsf{uart0.wellform}\ ps \wedge$$
$$\mathsf{uart\_has\_irq}\ \mathsf{IRQ}_{\mathrm{THRE}}\ ps \Rightarrow$$
$$\neg\mathsf{uart\_has\_irq}\ \mathsf{IRQ}_{\mathrm{THRE}}\ (\mathsf{SND}\ (\mathsf{SND}\ (\mathsf{uart0.mapped\_read}\ addr\ ps))) \qquad \text{(iv)}$$

Figure 5.1: Handling of different interrupt sources in the serial port model

```
13c: sub lr, lr, #4
140: push {r0, r1, r2, r3,
        r4, r5, r6, r7, r8,
        r9, sl, fp, ip, lr}
144: mrs r1, SPSR
148: push {r1}
14c: ldr r2, [pc, #276]
150: ldrh r0, [r2]
154: ldr r2, [pc, #272]
158: ldr ip, [r2]
15c: ldr r2, [pc, #268]
160: ldr r3, [pc, #268]
164: ldrh r7, [r2]
168: ldr r2, [pc, #264]
16c: ldrh r3, [r3]
170: ldrh r6, [r2]
174: ldr r1, [pc, #256]
178: ldr r5, [pc, #256]
17c: ldr r4, [pc, #256]
180: b 238 <uart0ISR+0xfc>
184: and r2, r2, #14
188: sub r2, r2, #2
18c: cmp r2, #10
190: ldrls pc, [pc, r2, lsl #2]
194: b 228 <uart0ISR+0xec>
198: .word 0x00000218
19c: .word 0x00000228
1a0: .word 0x000001d0
1a4: .word 0x00000228
1a8: .word 0x000001c4
1ac: .word 0x00000228
1b0: .word 0x00000228
1b4: .word 0x00000228
1b8: .word 0x00000228
1bc: .word 0x00000228
1c0: .word 0x000001d0
1c4: ldrb r2, [r1, #20]
1c8: b 230 <uart0ISR+0xf4>
1cc: mov r3, r2
1d0: ldrb r2, [r1]
1d4: strb r2, [r5, r3]
1d8: add r2, r3, #1
1dc: and r2, r2, #63
1e0: ldrb r8, [r1, #20]
1e4: cmp r2, r6
1e8: moveq r2, r3
1ec: tst r8, #1
1f0: bne 1cc <uart0ISR+0x90>
1f4: b 234 <uart0ISR+0xf8>
1f8: cmp r7, r0
1fc: moveq r2, r3
200: moveq ip, #0
204: beq 234 <uart0ISR+0xf8>
208: ldrb r2, [r4, r0]
20c: add r0, r0, #1
210: strb r2, [r1]
214: and r0, r0, #127
218: ldrb r2, [r1, #20]
21c: tst r2, #32
220: bne 1f8 <uart0ISR+0xbc>
224: b 230 <uart0ISR+0xf4>
228: ldrb r2, [r1, #20]
22c: ldrb r2, [r1]
230: mov r2, r3
234: mov r3, r2
238: ldrb r2, [r1, #8]
23c: tst r2, #1
240: beq 184 <uart0ISR+0x48>
244: ldr r2, [pc, #40]
248: strh r3, [r2]
24c: ldr r3, [pc, #20]
250: strh r0, [r3]
254: ldr r3, [pc, #16]
258: str ip, [r3]
25c: pop {r1}
260: msr SPSR_c, r1
264: ldm sp!, {r0, r1, r2,
        r3, r4, r5, r6, r7, r8,
        r9, sl, fp, ip, pc}^
268: .word 0x4000019c
26c: .word 0x400001a4
270: .word 0x40000092
274: .word 0x40000194
278: .word 0x400001ea
27c: .word 0xe000c000
280: .word 0x400001a8
284: .word 0x40000010
```

Figure 5.2: ARM assembly code for the serial port ISR

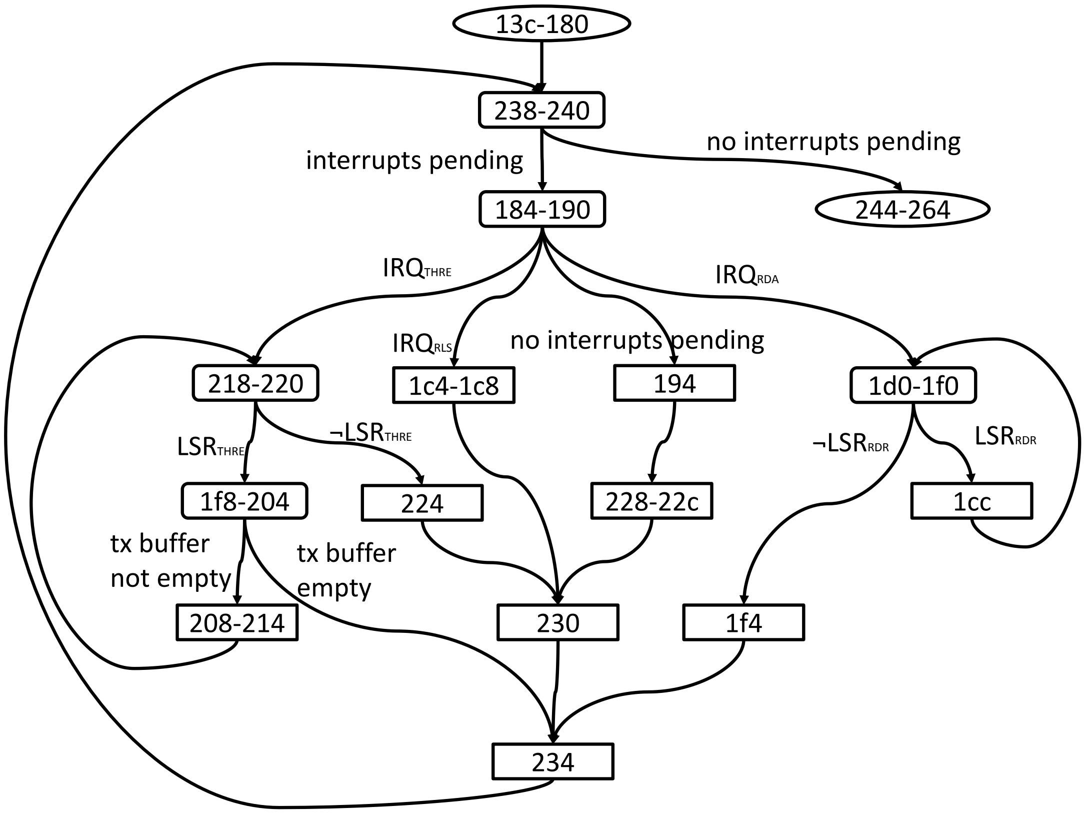

Figure 5.3: CFG of the ISR. Oval shape indicates the entry or exit block. Round rectangle shape indicates branches.

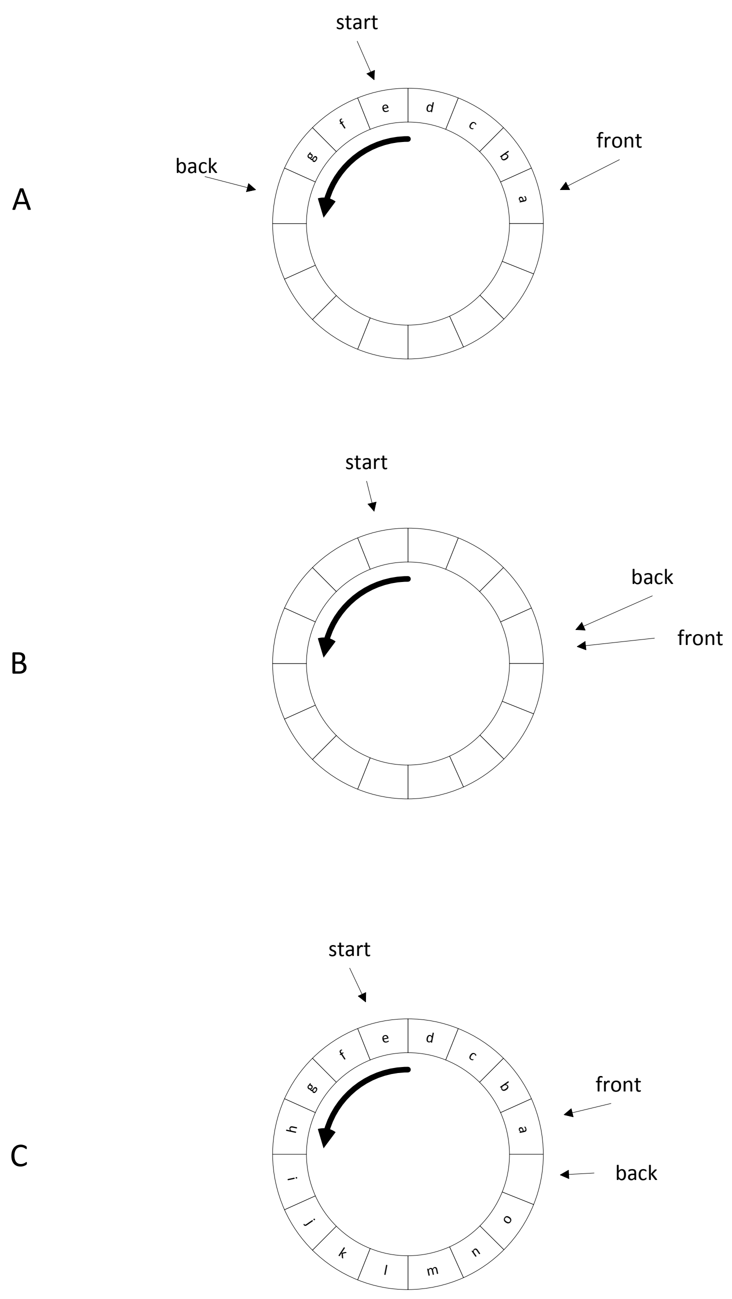

Figure 5.4: Circular buffer. The figure shows a circular buffer with size 16 at different stages of operation. In A, a list of [a;b;c;d;e;f;g] is stored in the buffer. In B, the buffer is empty. In C, the buffer is full, yet 1 slot is not used. The arced arrow indicates the growth of array index.

$$
\begin{array}{l}
\mathsf{circBuf}\ \mathit{start}\ \mathit{size}\ \mathit{df}\ f\ \mathit{front}\ \mathit{back}\ \mathit{str} \triangleq \\
1 <_{+} \mathit{size} \wedge \mathit{front} <_{+} \mathit{size} \wedge \mathit{back} <_{+} \mathit{size} \wedge (\forall\, i.\ i <_{+} \mathit{size} \Rightarrow \mathit{start} + i \in \mathit{df}) \wedge \\
\mathsf{case}\ \mathit{str}\ \mathsf{of} \\
\quad [\,] \rightarrow (\mathit{front} = \mathit{back}) \\
\ \|\ h :: \mathit{strtl} \rightarrow \mathit{back} \neq \mathit{front} \wedge (f(\mathit{start} + \mathit{front}) = h) \wedge \\
\quad\ \mathsf{circBuf}\ \mathit{start}\ \mathit{size}\ \mathit{df}\ f\ (\mathit{front} +_{\mathit{size}} 1)\ \mathit{back}\ \mathit{strtl}.
\end{array}
$$

Figure 5.5: circBuf definition

$$\vdash \mathsf{circBuf}\ \mathit{start}\ \mathit{size}\ \mathit{df}\ f\ \mathit{front}\ \mathit{back}\ \mathit{str} \Rightarrow$$
$$\forall\, i. i < \mathsf{LENGTH}\ \mathit{str} \Rightarrow$$
$$(\mathsf{EL}\ i\ \mathit{str} =$$
$$f\,(\mathit{start} + \mathsf{if}\ \mathsf{n2w}\ i <_{+} \mathit{size} - \mathit{front}\ \mathsf{then}\ \mathit{front} + \mathsf{n2w}\ i\ \mathsf{else}\ (\mathsf{n2w}\ i - (\mathit{size} - \mathit{front}))) \quad \text{(i)}$$
$$\vdash \mathsf{circBuf}\ \mathit{start}\ \mathit{size}\ \mathit{df}\ f\ \mathit{front}\ \mathit{back}\ \mathit{str} \Rightarrow$$
$$(\mathsf{LENGTH}\ \mathit{str} =$$
$$\mathsf{w2n}\ (\mathsf{if}\ \mathit{front} \leq_{+} \mathit{back}\ \mathsf{then}\ \mathit{back} - \mathit{front}\ \mathsf{else}\ \mathit{size} - (\mathit{front} - \mathit{back}))) \quad \text{(ii)}$$
$$\vdash \mathsf{circBuf}\ \mathit{start}\ \mathit{size}\ \mathit{df}\ f\ \mathit{front}\ \mathit{back}\ \mathit{str} \Rightarrow ((\mathit{front} = \mathit{back}) \iff (\mathit{str} = [])) \quad \text{(iii)}$$
$$\vdash \mathsf{circBuf}\ \mathit{start}\ \mathit{size}\ \mathit{df}\ f\ \mathit{front}\ \mathit{back}\ \mathit{str} \Rightarrow \mathsf{LENGTH}\ \mathit{str} < \mathsf{w2n}\ \mathit{size} \quad \text{(iv)}$$
$$\vdash \mathsf{circBuf}\ \mathit{start}\ \mathit{size}\ \mathit{df}\ f\ \mathit{front}\ \mathit{back}\ \mathit{str} \Rightarrow$$
$$(((\mathit{back} +_{\mathit{size}} 1)\ \mathit{size} = \mathit{front}) \iff (\mathsf{LENGTH}\ \mathit{str} = \mathsf{w2n}\ (\mathit{size} - 1))) \quad \text{(v)}$$
$$\vdash \mathsf{circBuf}\ \mathit{start}\ \mathit{size}\ \mathit{df}\ f\ \mathit{front}\ \mathit{back}\ (h :: \mathit{str}) \Rightarrow$$
$$\mathsf{circBuf}\ \mathit{start}\ \mathit{size}\ \mathit{df}\ f\ (\mathit{front} +_{\mathit{size}} 1)\ \mathit{back}\ \mathit{str} \quad \text{(vi)}$$
$$\vdash (\mathit{back} +_{\mathit{size}} 1) \neq \mathit{front} \wedge \mathsf{circBuf}\ \mathit{start}\ \mathit{size}\ \mathit{df}\ f\ \mathit{front}\ \mathit{back}\ \mathit{str} \Rightarrow$$
$$\mathsf{circBuf}\ \mathit{start}\ \mathit{size}\ \mathit{df}\ (((\mathit{start} + \mathit{back}) \mapsto \mathit{ch})\ f)\ \mathit{front}\ (\mathit{back} +_{\mathit{size}} 1)\ (\mathit{str}{+\!+}[\mathit{ch}]) \quad \text{(vii)}$$
$$\vdash \mathsf{circBuf}\ \mathit{start}\ \mathit{size}\ \mathit{df}\ f\ \mathit{front}\ \mathit{back}\ \mathit{str} \Rightarrow$$
$$\mathsf{circBuf}\ \mathit{start}\ \mathit{size}\ \mathit{df}\ (((\mathit{start} + \mathit{back}) \mapsto \mathit{ch})\ f)\ \mathit{front}\ \mathit{back}\ \mathit{str} \quad \text{(viii)}$$
$$\vdash (\mathit{back} +_{\mathit{size}} 1) \neq \mathit{front} \wedge \mathsf{circBuf}\ \mathit{start}\ \mathit{size}\ \mathit{df}\ f\ \mathit{front}\ \mathit{back}\ \mathit{str} \Rightarrow$$
$$\mathsf{circBuf}\ \mathit{start}\ \mathit{size}\ \mathit{df}\ f\ \mathit{front}\ (\mathit{back} +_{\mathit{size}} 1)\ (\mathit{str}{+\!+}[f\,(\mathit{start} + \mathit{back})] \quad \text{(ix)}$$

Figure 5.6: Circular buffer properties

$$
\begin{array}{ll}
\mathsf{aU}\ \mathit{divisor}\ \mathit{cushion}\ \mathit{cushion}_{\text{min}}\ \mathit{strm}_{\text{rx0}}\ \mathit{str}_{\text{rxbuf0}}\ \mathit{str}_{\text{tx}}\ \mathit{str}_{\text{txout}}\ \mathit{str}_{\text{txbuf}}\ \mathit{str}_{\text{rx}}\ \mathit{str}_{\text{rxin}} & \\
\mathit{str}_{\text{rxbuf}}\ \mathit{start}_{\text{rx}}\ \mathit{size}_{\text{rx}}\ \mathit{start}_{\text{tx}}\ \mathit{size}_{\text{tx}}\ \mathit{thre}\ \mathit{ie}_{\text{rls}}\ \mathit{ie}_{\text{rda}}\ \mathit{ie}_{\text{thre}}\ \mathit{irqs}\ \mathit{df}_{\text{rx}}\ \mathit{f}_{\text{rx}}\ \mathit{df}_{\text{tx}}\ \mathit{f}_{\text{tx}} & \\
\mathit{front}_{\text{rx}}\ \mathit{back}_{\text{rx}}\ \mathit{front}_{\text{tx}}\ \mathit{back}_{\text{tx}}\ \mathit{ps} \triangleq & \\
\mathsf{aBYTE\_MEMORY}\ \mathit{df}_{\text{rx}}\ \mathit{f}_{\text{rx}} * \mathsf{aBYTE\_MEMORY}\ \mathit{df}_{\text{tx}}\ \mathit{f}_{\text{tx}} * \mathit{aP}\ \mathit{ps} * & \text{(i)} \\
\left.\begin{array}{l}
\mathsf{cond}((\mathsf{get\_divisor}\ \mathit{ps} = \mathit{divisor}) \wedge (\mathsf{uart\_pending\_irqs}\ \mathit{ps} = \mathit{irqs}) \wedge \\
(\mathit{ps}.\mathsf{IER}_{\text{RLS}} = \mathit{ie}_{\text{rls}}) \wedge (\mathit{ps}.\mathsf{IER}_{\text{RDA}} = \mathit{ie}_{\text{rda}}) \wedge (\mathit{ps}.\mathsf{IER}_{\text{THRE}} = \mathit{ie}_{\text{thre}}) \wedge \\
(\mathit{ps}.\mathsf{LSR}_{\text{THRE}} = \mathit{thre}) \wedge
\end{array}\right\} & \text{(ii)} \\
\mathsf{uart0.wellform}\ \mathit{ps} \wedge \neg \mathit{ps}.\mathsf{LCR}_{\text{DLAB}} \wedge & \text{(iii)} \\
\mathit{ps}.\mathsf{clock} + \mathit{cushion}_{\text{min}} + \mathit{cushion} < \mathsf{get\_divisor}\ \mathit{ps} \wedge & \text{(iv)} \\
\left.\begin{array}{l}
\mathsf{sentString}\ \mathit{str}_{\text{txout}}\ \mathit{ps} \wedge (\mathsf{REVERSE}\ \mathit{str}_{\text{txbuf}} \mathsf{++} \mathit{str}_{\text{txout}} = \mathit{str}_{\text{tx}}) \wedge \\
\mathsf{circBuf}\ \mathit{start}_{\text{tx}}\ \mathit{size}_{\text{tx}}\ \mathit{df}_{\text{tx}}\ \mathit{f}_{\text{tx}}\ \mathit{front}_{\text{tx}}\ \mathit{back}_{\text{tx}}\ \mathit{str}_{\text{txbuf}} \wedge
\end{array}\right\} & \text{(v)} \\
\left.\begin{array}{l}
\mathsf{inputString}\ \mathit{str}_{\text{rxin}}\ \mathit{ps} \wedge (\mathit{str}_{\text{rxbuf}} \mathsf{++} \mathit{str}_{\text{rxin}} = \mathit{str}_{\text{rx}}) \wedge \\
\mathsf{circBuf}\ \mathit{start}_{\text{rx}}\ \mathit{size}_{\text{rx}}\ \mathit{df}_{\text{rx}}\ \mathit{f}_{\text{rx}}\ \mathit{front}_{\text{rx}}\ \mathit{back}_{\text{rx}}\ \mathit{str}_{\text{rxbuf}} \wedge \\
(\mathit{ps}.\mathsf{LSR}_{\text{RDR}} \Rightarrow \neg \mathsf{NULL} \mathit{str}_{\text{rxin}}) \wedge \mathsf{ishifted}\ \mathit{strm}_{\text{rx0}}\ \mathit{str}_{\text{rxbuf}}\ \mathit{ps}\ \mathit{str}_{\text{rxbuf}})
\end{array}\right\} & \text{(vi)}
\end{array}
$$

Figure 5.7: aU definition

$\vdash \mathsf{aU}\ \mathit{divisor}\ \mathit{cushion}_{\mathrm{min}}\ (\mathit{cushion} + \mathit{offset})\ \mathit{strm}_{\mathrm{rx0}}\ \mathit{str}_{\mathrm{rxbuf}}\ \mathit{str}_{\mathrm{tx}}\ \mathit{str}_{\mathrm{txout}}\ \mathit{str}_{\mathrm{txbuf}}$
$\mathit{str}_{\mathrm{rx}}\ \mathit{str}_{\mathrm{rxin}}\ \mathit{str}_{\mathrm{rxbuf2}}\ \mathit{start}_{\mathrm{rx}}\ \mathit{size}_{\mathrm{rx}}\ \mathit{start}_{\mathrm{tx}}\ \mathit{size}_{\mathrm{tx}}\ \mathit{thre}\ \mathit{ie}_{\mathrm{rls}}\ \mathit{ie}_{\mathrm{rda}}\ \mathit{ie}_{\mathrm{thre}}\ \mathit{irqs}\ \mathit{df}_{\mathrm{rx}}\ f_{\mathrm{rx}}$
$\mathit{df}_{\mathrm{tx}}\ f_{\mathrm{tx}}\ \mathit{front}_{\mathrm{rx}}\ \mathit{back}_{\mathrm{rx}}\ \mathit{front}_{\mathrm{tx}}\ \mathit{back}_{\mathrm{tx}}\ \mathit{ps} =$
$\mathsf{aU}\ \mathit{divisor}\ (\mathit{cushion}_{\mathrm{min}} + \mathit{offset})\ \mathit{cushion}\ \mathit{strm}_{\mathrm{rx0}}\ \mathit{str}_{\mathrm{rxbuf0}}\ \mathit{str}_{\mathrm{tx}}\ \mathit{str}_{\mathrm{txout}}\ \mathit{str}_{\mathrm{txbuf}}$
$\mathit{str}_{\mathrm{rx}}\ \mathit{str}_{\mathrm{rxin}}\ \mathit{str}_{\mathrm{rxbuf}}\ \mathit{start}_{\mathrm{rx}}\ \mathit{size}_{\mathrm{rx}}\ \mathit{start}_{\mathrm{tx}}\ \mathit{size}_{\mathrm{tx}}\ \mathit{thre}\ \mathit{ie}_{\mathrm{rls}}\ \mathit{ie}_{\mathrm{rda}}\ \mathit{ie}_{\mathrm{thre}}\ \mathit{irqs}\ \mathit{df}_{\mathrm{rx}}\ f_{\mathrm{rx}}$
$\mathit{df}_{\mathrm{tx}}\ f_{\mathrm{tx}}\ \mathit{front}_{\mathrm{rx}}\ \mathit{back}_{\mathrm{rx}}\ \mathit{front}_{\mathrm{tx}}\ \mathit{back}_{\mathrm{tx}}\ \mathit{ps}$

$\vdash \neg\mathsf{aU}\ \mathit{divisor}\ \mathit{cushion}_{\mathrm{min}}\ (\mathit{cushion} + \mathit{offset})\ \mathit{strm}_{\mathrm{rx0}}\ \mathit{str}_{\mathrm{rxbuf0}}\ \mathit{str}_{\mathrm{tx}}\ \mathit{str}_{\mathrm{txout}}$
$\mathit{str}_{\mathrm{txbuf}}\ \mathit{str}_{\mathrm{rx}}\ \mathit{str}_{\mathrm{rxin}}\ \mathit{str}_{\mathrm{rxbuf}}\ \mathit{start}_{\mathrm{rx}}\ \mathit{size}_{\mathrm{rx}}\ \mathit{start}_{\mathrm{tx}}\ \mathit{size}_{\mathrm{tx}}\ \mathit{thre}\ \mathit{ie}_{\mathrm{rls}}\ \mathit{ie}_{\mathrm{rda}}\ \mathit{ie}_{\mathrm{thre}}\ \mathit{irqs}$
$\mathit{df}_{\mathrm{rx}}\ f_{\mathrm{rx}}\ \mathit{df}_{\mathrm{tx}}\ f_{\mathrm{tx}}\ \mathit{front}_{\mathrm{rx}}\ \mathit{back}_{\mathrm{rx}}\ \mathit{front}_{\mathrm{tx}}\ \mathit{back}_{\mathrm{tx}}\ =$
$\neg\mathsf{aU}\ \mathit{divisor}\ (\mathit{cushion}_{\mathrm{min}} + \mathit{offset})\ \mathit{cushion}\ \mathit{strm}_{\mathrm{rx0}}\ \mathit{str}_{\mathrm{rxbuf0}}\ \mathit{str}_{\mathrm{tx}}\ \mathit{str}_{\mathrm{txout}}\ \mathit{str}_{\mathrm{txbuf}}$
$\mathit{str}_{\mathrm{rx}}\ \mathit{str}_{\mathrm{rxin}}\ \mathit{str}_{\mathrm{rxbuf}}\ \mathit{start}_{\mathrm{rx}}\ \mathit{size}_{\mathrm{rx}}\ \mathit{start}_{\mathrm{tx}}\ \mathit{size}_{\mathrm{tx}}\ \mathit{thre}\ \mathit{ie}_{\mathrm{rls}}\ \mathit{ie}_{\mathrm{rda}}\ \mathit{ie}_{\mathrm{thre}}\ \mathit{irqs}\ \mathit{df}_{\mathrm{rx}}$
$f_{\mathrm{rx}}\ \mathit{df}_{\mathrm{tx}}\ f_{\mathrm{tx}}\ \mathit{front}_{\mathrm{rx}}\ \mathit{back}_{\mathrm{rx}}\ \mathit{front}_{\mathrm{tx}}\ \mathit{back}_{\mathrm{tx}}$

Figure 5.8: Theorems showing accumulation of the minimum cushion time

$$\begin{array}{l}\vdash \mathit{cushion}_{\mathrm{min1}} \geq \mathit{cushion}_{\mathrm{min2}} \Rightarrow \\ \quad (\mathsf{aU}\ \mathit{divisor}\ \mathit{cushion}\ \mathit{cushion}_{\mathrm{min1}}\ \mathit{strm}_{\mathrm{rx0}}\ \mathit{str}_{\mathrm{rxbuf0}}\ \mathit{str}_{\mathrm{tx}}\ \mathit{str}_{\mathrm{txout}}\ \mathit{str}_{\mathrm{txbuf}}\ \mathit{str}_{\mathrm{rx}}\ \mathit{str}_{\mathrm{rxin}} \\ \quad \mathit{str}_{\mathrm{rxbuf}}\ \mathit{start}_{\mathrm{rx}}\ \mathit{size}_{\mathrm{rx}}\ \mathit{start}_{\mathrm{tx}}\ \mathit{size}_{\mathrm{tx}}\ \mathit{thre}\ \mathit{ie}_{\mathrm{rls}}\ \mathit{ie}_{\mathrm{rda}}\ \mathit{ie}_{\mathrm{thre}}\ \mathit{irqs}\ \mathit{df}_{\mathrm{rx}}\ f_{\mathrm{rx}}\ \mathit{df}_{\mathrm{tx}}\ f_{\mathrm{tx}}\ \mathit{front}_{\mathrm{rx}} \\ \quad \mathit{back}_{\mathrm{rx}}\ \mathit{front}_{\mathrm{tx}}\ \mathit{back}_{\mathrm{tx}}\ \mathit{ps}) \triangleright \\ \quad (\mathsf{aU}\ \mathit{divisor}\ \mathit{cushion}\ \mathit{cushion}_{\mathrm{min2}}\ \mathit{strm}_{\mathrm{rx0}}\ \mathit{str}_{\mathrm{rxbuf0}}\ \mathit{str}_{\mathrm{tx}}\ \mathit{str}_{\mathrm{txout}}\ \mathit{str}_{\mathrm{txbuf}}\ \mathit{str}_{\mathrm{rx}}\ \mathit{str}_{\mathrm{rxin}} \\ \quad \mathit{str}_{\mathrm{rxbuf}}\ \mathit{start}_{\mathrm{rx}}\ \mathit{size}_{\mathrm{rx}}\ \mathit{start}_{\mathrm{tx}}\ \mathit{size}_{\mathrm{tx}}\ \mathit{thre}\ \mathit{ie}_{\mathrm{rls}}\ \mathit{ie}_{\mathrm{rda}}\ \mathit{ie}_{\mathrm{thre}}\ \mathit{irqs}\ \mathit{df}_{\mathrm{rx}}\ f_{\mathrm{rx}}\ \mathit{df}_{\mathrm{tx}}\ f_{\mathrm{tx}}\ \mathit{front}_{\mathrm{rx}} \\ \quad \mathit{back}_{\mathrm{rx}}\ \mathit{front}_{\mathrm{tx}}\ \mathit{back}_{\mathrm{tx}}\ \mathit{ps})\end{array}$$

Figure 5.9: Monotonicity of the cushion time

```
sample $str_{\mathrm{buf}}$ $strm$ $strm_0$ $\triangleq$
case $str_{\mathrm{buf}}$ of
    $[] \rightarrow \exists\, m. strm =$ cutStream $m$ $strm_0$
 $\|$ $(h :: tl) \rightarrow$
    $\exists\, n.$ $(strm_0\ n =$ SOME $h) \wedge$ sample $tl$ $strm$ (cutStream $(n+1)$ $strm_0$)
ishifted $strm_{\mathrm{rx0}}$ $str_{\mathrm{rxbuf0}}$ $ps$ $str_{\mathrm{rxbuf}}$ $\triangleq$
$\exists\, n.$
$n\ \leq$ LENGTH $str_{\mathrm{rxbuf}}$ $\wedge$ (TAKE $n$ $str_{\mathrm{rxbuf}} = str_{\mathrm{rxbuf0}}) \wedge$
if $ps$.LSR$_{\mathrm{RDR}}$
then sample(DROP $n$ $str_{\mathrm{rxbuf}}$ ++[$ps$.RBR])$ps$.in $strm_{\mathrm{rx0}}$
else sample (DROP $n$ $str_{\mathrm{rxbuf}}$) $ps$.in $strm_{\mathrm{rx0}}$
```

Figure 5.10: Definitions of sample and ishifted

Table 5.3: Placeholders used in the Hoare triple for the ISR

constant	description
$start_{\mathrm{rx}}$	0x400001A8
$size_{\mathrm{rx}}$	64
$start_{\mathrm{tx}}$	0x40000010
$size_{\mathrm{tx}}$	128
$front\_addr_{\mathrm{rx}}$	0x400001EA
$back\_addr_{\mathrm{rx}}$	0x40000194
$front\_addr_{\mathrm{tx}}$	0x4000019C
$back\_addr_{\mathrm{tx}}$	0x40000092
$front_{\mathrm{rx}}$	$f \nearrow_2^{16}$ 0x400001EA
$back_{\mathrm{rx}}$	$f \nearrow_2^{16}$ 0x40000194
$front_{\mathrm{tx}}$	$f \nearrow_2^{16}$ 0x4000019C
$back_{\mathrm{tx}}$	$f \nearrow_2^{16}$ 0x40000092
$newf$	f updated with all the variables in its domain

```
latency $irqs \triangleq$
case $irqs$ of
   $[] \rightarrow 0$
 $\|$ $(\mathsf{IIR}_{\mathrm{THRE}} :: tl) \rightarrow 24 + \mathsf{latency}\ tl$
 $\|$ $(\mathsf{IIR}_{\mathrm{RDA}} :: tl) \rightarrow 19 + \mathsf{latency}\ tl$
 $\|$ $(\mathsf{IIR}_{\mathrm{RLS}} :: tl) \rightarrow 12 + \mathsf{latency}\ tl$
```

Figure 5.11: Definition of lantency

CHAPTER 6

CORRECTNESS PROOF FOR INTERRUPT-DRIVEN SERIAL PORT DRIVER

Interrupts present concurrency challenges. At any time when an instruction in the main program is executed, it is possible that an ISR may execute due to a hardware interrupt request. Its effect includes modifying memory content, changing the device state, and causing delay to the execution of the main program and handling of other interrupts. In most cases, the values in processor registers are restored. To reason about the main program, the effect of the ISRs installed in the system must be accounted for.

My approach is to describe the semantics of this instruction with the possible effect of the ISRs included. After obtaining such semantics for each instruction of the main program, traditional techniques used to reason about sequential programs can be used to reason about the main program.

In most cases, the effect of the ISRs is transparent. This is because the memory of the main program is not accessed by the ISR.

For a main program that has resources shared with an ISR, there is an invariant maintained between them in most situations. Most of the time, the main program can execute without disabling the interrupts, while maintaining this invariant. However, if the only thing which can be reasoned about in this scenario is just an invariant, it is not very useful for verifying deep properties. Interesting changes happen at program points when the old invariant is broken, and the new one is established. This usually occurs inside an atomic section.

The main program runs in a spectrum of different modes. At one end is interrupt mode when interrupt handling is enabled on the processor, and all the hardware interrupts are enabled. In this case only an invariant can be reasoned about on the shared resources.

At the other end is atomic mode when interrupt handling is disabled on the processor. In this mode significant changes are made so that the old invariant would not hold inside atomic section. But it is not hard to verify since there is no concurrency.

In between are the selective atomic modes when some of the hardware interrupt sources are disabled while interrupt handling is enabled on the processor. Disabling one interrupt source will remove some resources from the shared resource set. For these resources, the program works as in atomic mode. The invariant involving only these resources can be broken in the main program, and a new one can be established. For other resources, the program works just as in interrupt mode. Disabling all the interrupts is equivalent to atomic mode.

6.1 Serial Port Driver

I verified function uart0Putch. Its code is listed in Figure 6.1. In the process, I also verified two assembly routines disableIRQ and restoreIRQ. Their code is listed in Figure 6.2. uart0Putch tries to write a byte to the tx pipeline if the tx buffer has room. When the flag *tx_running* is set, it writes to the tx buffer in memory. Otherwise, it writes directly to THR on the serial port device.

tx_running is necessary because $\mathsf{IIR}_{\mathrm{THRE}}$ is only set after THR is emptied. If THR remains empty, $\mathsf{IIR}_{\mathrm{THRE}}$ is not set. This behavior is reasonable. Otherwise, interrupt $\mathsf{IRQ}_{\mathrm{THRE}}$ will persistently fire even if there is no sending activity going on. So to trigger interrupt $\mathsf{IRQ}_{\mathrm{THRE}}$, THR must be written first. *tx_running* is set when THR is written to in uart0Putch. In the ISR, *tx_running* is cleared when both the tx buffer and THR are emptied.

The structure of the code is rather simple. In c80, it saves registers to the stack. Between c84 and ca0, it reads $\mathit{front}_{\mathrm{tx}}$ and $\mathit{back}_{\mathrm{tx}}$, and decides if there is any room in the tx buffer. In this process $\mathsf{IRQ}_{\mathrm{THRE}}$ is not disabled, the ISR can modify $\mathit{front}_{\mathrm{tx}}$ concurrently. The room calculated here might be out of date. However, the code is correct in the sense that

1. If it is deemed that there is room available, then surely there is room available in the tx buffer, and it is safe to proceed to write to the tx pipeline.
2. If it is deemed that there is no room available, then there may or may not be room available in the tx buffer. Whatever the case is, it is always safe to quit.

Between ca4 and ca8 is the exit branch under the condition that there is no room in the tx buffer.

Between cac and cb0, uart0Putch calls procedure disableIRQ to disable interrupt handling on the processor. From cb4 to cbc it disables hardware interrupt $\mathsf{IRQ}_{\mathrm{THRE}}$ by clearing $\mathsf{IER}_{\mathrm{THRE}}$ on the serial port. Then in cc0 it calls procedure restoreIRQ to restore interrupt handling on the processor. Atomic mode starts inside disableIRQ and ends inside restoreIRQ.

Then the execution enters selective atomic mode. I call it THRE-atomic mode. From cc4 to ccc uart0Putch reads *tx_running* to see if it is set. Then two branches are implemented using the conditional execution between cd0 and cec. The instructions ending with eq implement the branch with the condition that *tx_running* is not set. In this branch it sets *tx_running* and writes the byte to THR directly. The instructions ending with neq implement the branch with the condition that *tx_running* is set. In this branch uart0Putch appends the byte to the tx buffer and updates $back_{\mathrm{tx}}$.

In cf0 uart0Putch calls procedure disableIRQ again to disable interrupt handling on the processor. THRE-atomic mode ends inside disableIRQ, and atomic mode begins. Between cf4 and d00 hardware interrupt $\mathsf{IRQ}_{\mathrm{THRE}}$ on the serial port is enabled by setting $\mathsf{IER}_{\mathrm{THRE}}$ on the serial port. Then in d04 procedure restoreIRQ is called to restore interrupt handling on the processor. Again, atomic mode ends inside restoreIRQ, at the same time interrupt mode resumes.

The final two instructions d08 and d0c prepare the return value, restore the registers and return.

In summary, uart0Putch goes through three different modes in execution:

1. Interrupt mode. In this mode interrupt handling is enabled and no hardware interrupts are disabled. All instructions before cb0 or after d04 are in this mode.
2. Atomic mode. In this mode interrupts handling are disabled. Al instructions between cb0 and cc0, or between cf0 and d04 are in this mode. $\mathsf{IER}_{\mathrm{THRE}}$ is

cleared and set, respectively, in these two blocks.

3. THRE-atomic mode. In this mode interrupt handling is enabled on the processor, but $\mathsf{IER}_{\mathrm{THRE}}$ is cleared. Interrupt $\mathsf{IRQ}_{\mathrm{THRE}}$ request does not occur. All instructions between cc4 and cec are in this mode.

Instructions in procedures disableIRQ and restoreIRQ (see Figure 6.2) can be partitioned in a similar way. However, the partition differs between two calling sites. For example, at calling site cb0, disableIRQ enters in interrupt mode, and exits in atomic mode. At calling site cf0, disableIRQ enters in THRE-atomic mode, and exits in atomic mode.

6.2 Assertion Formula for the ISR

Previously, in Chapter 5, I proved the correctness of the serial port ISR as a standalone function. The Hoare triple for the ISR is shown in Equation 5.4. It is a strong result in the sense that it is not an invariant. To integrate its effect into the semantics of the instructions of the main program uart0Putch, the assertion formula defined in Figure 5.7 needs to be revised so that the revised Hoare triple of the ISR should describe some invariants observed in the main program.

6.2.1 Interrupt Mode

The formula used to assert ISR in interrupt mode is shown in Figure 6.3. It expands aU assertion in Figure 5.7 in two ways.

Inside the ISR, only the storage arrays of the rx and tx buffers are in memory, while *front* and *back* are in registers. *tx_running* flag is also in register. From the perspective of the main program, all these are in memory. So a memory region assertion (ii) is added. This memory region is for *front* and *back* of the rx and tx buffers and *tx_running* flag, as shown in (iii) and (iv) .

There are two additional predicates:

1. At (v) it is asserted that the room in the tx buffer is larger than an initial value $space_{\mathrm{tx0}}$. This is an invariant. It is useful to verify the block where benign race around $front_{\mathrm{tx}}$ occurs.
2. At (vi) the consistency about the *tx_running* flag is also built in. It asserts that when *tx_running* is not set, the tx pipeline including the tx buffer and THR is

empty. Once the serial port enters this state, interrupt $\mathsf{IIR}_{\mathrm{THRE}}$ request does not occur unless a byte is written into THR.

6.2.2 THRE-Atomic Mode

In THRE-atomic mode, $\mathsf{IER}_{\mathrm{thre}}$ is disabled, another formula in Figure 6.4 is used to assert the ISR. It is based on Figure 6.3 with one new predicate (vii) to reflect that $\mathsf{IRQ}_{\mathrm{THRE}}$ is excluded from pending interrupts.

6.3 Assertion Formula for uart0Putch

Generally, different formulas are used to assert uart0Putch in different modes. This is because the resource conflict varies between different modes. However, interrupt mode and atomic mode share the same formula. Theoretically, in atomic mode there is no resource conflict. One does not need to hide any resources and can achieve maximum accuracy in assertion. However, uart0Putch does nothing interesting in atomic mode, i.e., there are no changes to the rx and tx pipelines and buffers. The same assertion formula for interrupt mode works for atomic mode.

6.3.1 Interrupt and Atomic Modes

The formula used to assert uart0Putch in interrupt mode and atomic mode is shown in Figure 6.5. It differs from the one in Figure 6.3 only in the timing constraints. In the theorem in Equation 5.4, it is required that there must be enough cushion before the internal clock of the serial port resets. This ensures that the ISR terminates and that there is no delay in handling the interrupt request, i.e., no incoming data is dropped.

To discharge this condition at the entry of the ISR, two conditions are needed outside the ISR. They are built in the formula in Figure 6.5:

1. At (vi) it is asserted that the serial port must be slow enough.
2. At (vii) it is asserted that the interrupt latency for the serial port must be low enough.

6.3.2 THRE-Atomic Mode

The formula used to assert uart0Putch in THRE-atomic mode is shown in Figure 6.6. Similar to the formula in Figure 6.4, one new predicate (x) is added to reflect that $\mathsf{IRQ}_{\mathrm{THRE}}$ is excluded from pending interrupts.

6.4 Resource Conflicts and Assertion Weakening

All the assertion formula in Figures 6.3, 6.4, 6.5 and 6.6, share almost the same parameters. The ones for uart0Putch in Figures 6.5 and 6.6 do have one more parameter $divisor_{\mathrm{min}}$. These parameters come from the following four groups:

1. $\{ps, df_{\mathrm{rx}}, f_{\mathrm{rx}}, df_{\mathrm{tx}}, f_{\mathrm{tx}}, df_{\mathrm{g}}, f_{\mathrm{g}}\}$ are all actual low-level resources in the system. ps is the state of the serial port. There are three memory regions: (f_{rx}, df_{rx}) for rx buffer, (f_{tx}, df_{tx}) for tx buffer, and (f_{g}, df_{g}) for *front* and *back* of the buffers and tx_running flag.
2. $\{divisor, str_{\mathrm{tx}}, str_{\mathrm{txout}}, str_{\mathrm{txbuf}}, str_{\mathrm{rx}}, str_{\mathrm{rxin}}, str_{\mathrm{rxbuf}}, tx\_running, thre, ie_{\mathrm{rls}}, ie_{\mathrm{rda}}, ie_{\mathrm{thre}}, irqs, front_{\mathrm{rx}}, back_{\mathrm{rx}}, front_{\mathrm{tx}}, back_{\mathrm{tx}}\}$ provide a high-level view of the low-level resources. The correspondence between the high-level view and low-level resources is listed in Table 6.1. str_{rx} and str_{tx} include information from both the serial port state and memory.
3. $\{cushion_{\mathrm{min}}, cushion, divisor_{\mathrm{min}}, strm_{\mathrm{rx0}}, str_{\mathrm{rxbuf0}}, space_{\mathrm{tx0}}\}$ are parameters specified by the user. $divisor_{\mathrm{min}}$ is used to specify the lower bound of the clock divisor of the serial port. The speed of the serial port gets lower with higher value of $divisor_{\mathrm{min}}$. $space_{\mathrm{tx0}}$ is used to specify the lower bound of the space available in tx buffer. Others are discussed in Section 5.4.
4. $\{start_{\mathrm{rx}}, size_{\mathrm{rx}}, start_{\mathrm{tx}}, size_{\mathrm{tx}}, front\_addr_{\mathrm{rx}}, back\_addr_{\mathrm{rx}}, front\_addr_{\mathrm{tx}}, back\_addr_{\mathrm{tx}}, tx\_running\_addr\}$ are place holders for constants including start and size parameters as well as addresses of *front* and *back* of the rx and tx buffers and the address of *tx_running*.

The main program uart0Putch and the ISR share the following resources: the array of tx buffer, $front_{\mathrm{tx}}$, $front_{\mathrm{tx}}$, *tx_running* and THR. The details of the access conflicts in the driver are listed in Table 6.2 in terms of the high-level parameters. Side effects due to device access are also included in write accesses.

The formula in Figure 6.3, and Figures 6.4, 6.5 and 6.6 are abstracted into functions over their parameters. The order of these parameters matters, because it decides how the predicates can be weakened by hiding when application of these functions are used as separation assertions to assert the program. Only the last parameter is hidden in weakening.

In each mode, the parameters which are updated by the ISR should be hidden, because whether the ISR will execute is nondeterministic, and the specific values of the parameters cannot be decided. When a high-level parameter is hidden, the corresponding low-level ones need to be hidden too. So, I put all the parameters needing to be hidden into a tuple and put it as the last formal parameter of the function. All the parameters related to the rx function are always in this tuple.

Figure 6.7 shows the abstraction of the assertion formula for the serial port ISR. The formula in Figure 6.3 is abstracted into aU_1 as in (i), and the one in Figure 6.4 is abstracted into aU_2 or aU_3 as in (ii) and (iii).

Figure 6.8 shows the abstraction of the assertion formula for uart0Putch. The formula in Figure 6.5 is abstracted into aU_4 and aU_8 as in (i) and (v), and the one in Figure 6.6 is abstracted into aU_5, aU_6 and aU_7 as in (ii), (iii) and (iv).

These functions are used in the preconditions and postconditions of the Hoare triples in the proof:

1. aU_4 is used to assert the main code uart0Putch in interrupt mode. Correspondingly aU_1 is used in the ISR.
2. aU_4 is also used to assert the main code uart0Putch in atomic mode, because there is no concurrency, nothing needs to be hidden in this mode; any function could be used here.
3. aU_5 and aU_6 are used to assert uart0Putch in THRE-atomic mode. In this mode there are two branches. For $\mathit{tx\_running} = 0$ branch, thre value in the serial port state is only updated by the main program, so aU_6 is used. For $\mathit{tx\_running} \neq 0$ branch, thre value in the serial port state may be updated by the ISR, so aU_5 is used. Correspondingly aU_2 is used in the ISR.
4. After the branches are combined in uart0Putch in interrupt mode, aU_8 is used. Because the memory region for the tx buffer may or may not be modified, it needs to be hidden, so does $\mathit{back}_{\mathrm{tx}}$. Correspondingly aU_1 is used in the ISR. As a result, in the postcondition of the Hoare triple for the whole uart0Pucth function, aU_8 is used.

5. aU_7 is used to assert the code right after atomic section is ended in which $\mathsf{IER}_{\mathrm{THRE}}$ is disabled. However, there may be still outstanding interrupts which require handling. These interrupts may occur during atomic section. Correspondingly aU_3 is used for the ISR.

6.5 Effect of the ISR

The Hoare triple about the ISR proved in Chapter 5 is a strong one in the sense that it is not just an invariant. It describes the ideal behavior of the ISR. In this section I present the Hoare triples of the ISR not as a standalone program, but to be integrated into the Hoare triples of instructions in the main program.

6.5.1 Interrupt Mode

For interrupt mode, the Hoare triple in Equation 5.4 is adapted into the Hoare triple in Figure 6.9. Compared to the theorem in Equation 5.4, the assertion on the serial port related resource in Figure 6.9 is almost an invariant except the lower bound of the cushion time, which changes from 85 to 0 from the precondition to the postcondition. It is used to integrate the effect of ISR in interrupt mode where the main code is asserted using aU_4 or aU_8.

6.5.2 THRE-Atomic Mode

In THRE-atomic mode, aU_2 or aU_3 is used to describe the effect of the ISR. The theorem in Figure 6.10 shows the Hoare triple of the ISR is adapted using aU_2. It is used in conjunction with aU_5 in the main code.

6.6 Correctness of uart0Putch

The Hoare triple for the working branch of uart0Putch is shown in Figure 6.11. The details of the placeholders are shown in Table 6.3.

uart0Putch is a function. It follows the calling convention of the ARM architecture. $\mathsf{LR}_{\mathsf{svc}}$ is used to hold the return address, as shown in (ii) and (x). Registers 4 through 7 and the link register are saved to the stack in svc mode first and restored later, as shown in (vi), (xiv), (v), and (x). The hidden registers are intraprocedure scratch registers in ARM. R_0 holds the return value. Interrupt handling is enabled at the entry and at the exit, as shown in (iv) and (xii). Assertions to the registers in irq mode are exposed here, especially the stack size in the irq mode. It is shown in (iii) and (x).

The alignment requirement is shown in (viii) for both the svc mode and the irq mode. The code is the union of the doe from uart0Putch and the ISR.

aU_8 is used to assert the resource related to the serial port, including the tx and rx buffers, *tx_running* flag, and the state of the serial port device.

The precondition (vii) and the postcondition (vii) indeed show that the byte in R_0 at the entry, r_0, is appended to the tx pipeline. More specifically, it is appended to str_{tx}, which abstracts the string in the tx pipeline.

The values for $space_{\mathrm{tx0}}$ is 1 in the precondition, and 0 in the postcondition. This means if there is room available at the entry, it may be used up by the appending.

The value for $cushion_{\mathrm{min}}$ and $divisor_{\mathrm{min}}$ are both 97. $cushion_{\mathrm{min}}$ measures the minimum requirement of cushion time. It is the sum of the longest path from all atomic sections and the longest path in the ISR. This value ensures that serial port interrupts are always taken in time. $divisor_{\mathrm{min}}$ measures the minimum requirement for the clock divisor in the serial port device. It must not be smaller than $cushion_{\mathrm{min}}$. This value ensures that the serial port is slow enough that ISR will terminate.

6.7 Proof Methods

In this section I discuss some techniques from my experience in the proof process. Since no automatic tool is available, such techniques are valuable for other projects of a similar nature.

6.7.1 Assertion Propagation

Constants are propagated in sequential composition. In my approach, an additional pass of constant propagation is necessary before composition is attempted.

Among the constants propagated are the addresses. With this it can be determined if a ld/str instruction is accessing memory or a device. Thus, the impossible execution paths due to the confusion on memory and device access are trimmed early as possible.

The assertion formulas are also propagated. In most cases, the switch of the assertion function occurs at the edge of an atomic section or selective atomic section. The propagation of the assertion function works in most of time.

Another piece of information needed to be propagated is the branch condition in the conditional execution. The branch condition must be literally propagated so that two branches are not mixed together.

6.7.2 Inlining

The two procedures which disable and enable the processor's interrupt handling, disableIRQ and restoreIRQ are inlined in the proof. It may sound like a repetition of work, but it is necessary.

Firstly, each of these two procedures starts or ends atomic mode. Inlining helps to piece atomic sections with the ones in uart0Putch to form complete atomic sections.

Secondly, the modes are different at different calling sites. For example, when disableIRQ is first called, the program is in interrupt mode. When it is called the second time, the program is in THRE-atomic mode. So different assertion formulas have to be used. Inlining makes this straightforward. In short, the assertion is context-sensitive.

6.7.3 Conditional Execution

With the assertion propagation, the paths along the same branch can be discovered. Two branches are composed independently, then merged together. The reason is that two branches may use different assertion formulas. Merging at every instruction will either produce complicated if-then-else terms if precision is sought, or will lose precision if the two branches are weakened to a common ground. In short, the assertion is path sensitive.

6.7.4 Weakening and Live Variables

After integrating the effect of the ISR on the semantics of each instruction in the main program, we have a sequence of Hoare triples. They need to be composed to form the final Hoare triple about the main program. When composing two Hoare triples sequentially, often the postcondition of the first one needs be weakened, or the precondition of the second one needs be strengthened, or both.

Although a separation assertion can be weakened in many ways inside HOL, I use hiding for ease of mechanical manipulation. For a separation assertion in the precondition, its last variable can be hidden only if it does so to appear in the postcondition and other terms in the precondition. Using a resource will always produce the first hazard, while copying values around will create the second one. The nice thing is that for a separation assertion in the postcondition, its last variable can always be hidden.

A variable can be safely hidden form the postcondition if it is at the end of its life cycle. It is natural to start the weakening from the last Hoare triple in the sequence, because at the end of the function all variables are dead.

6.7.5 Automation

In the context of theorem proving, automation is opposite to interactive proof. Even though it is unrealistic to achieve full automation in the proof, it is crucial to have enough automation that the proof is manageable.

There are several characteristics in this work which make automation difficult, if not impossible. Complex control flows in the program make automation hard. For example, deriving the Hoare triple for a loop often needs inductive proof.

The introduction of device models makes automation hard. The complexity of the device model is at the root. For the serial port model, recursive definitions, if-then-else control, and MOD operation in the clock function all make automation hard.

High-level assertions also make automation hard. In the assertion formula used in this work, existential qualified variables and recursive definition make automation hard.

Model accuracy also comes at the cost of automation. In this work, both the ARM model from the University of Cambridge and the serial port model are constructed using fixed-width words as the underlying data types. Even though decision procedures for words in HOL4 are much improved [10, 35], some proof still needs interactive efforts.

Despite these issues, I still tried to achieve some automation which helps to make the project manageable. I adapted the scripts from the Cambridge model to automate the derivation of the semantics theorem for ARM instructions for the ARM SoC model.

I placed more emphasis on automation of the high-level work flow, such as the application of inference rules, shown in Figure 4.13. Integration of the ISR effect into the semantics of the ARM instructions for the main program is automated with the help of large amount of lemmas proved interactively. Sequential composition and branch combination both are automated.

These lemmas are proved interactively. This can be viewed as an abstraction process. For example, the serial port model is defined on a group of flags, registers and input and output streams. The lemmas proved for the serial port model describe the operation of the serial port model with each command from the processor or its own transition in terms of flags, the input and output strings. The lemmas use the same abstraction as the assertion formula, they can be used to automate the integration process.

On the main program side, access to the shared resources, such as device registers, buffers and flags are also lifted in terms of the high-level data structures such as circular buffers and input and output strings. By recognizing the operations on the high-level data structure by the machine code, I proved enough lemmas and use them to automate the integration process.

Besides proof, some aspects of my work can be helped by some machinery. For example, worst-case execution time analysis may be used to provide a start point for $\mathit{cushion}_{\min}$, which is the sum of the worst-case execution-time (WCET) for the longest atomic section and for the ISR. Control flow graph and the execution modes can be automatically generated using static analysis. There are lots of work on these analysis. Wilhelm et al. did an excellent survey of WCET analysis [99]. Diablo [22] can be used to extract control-flow graph of ARM assembly programs.

6.8 Benign Racing and Nonmonotonic Branch

The Hoare triple in Figure 6.11 is only for the working branch of uart0Putch, when there is room in the tx buffer at the entry point of putch. This condition is reflected in the cond term in Figure 6.11. However, I could not prove that uart0putch must take the exit branch under a condition, because the exit branch cannot be composed to the Hoare triple of the entry block. The branch condition for the exit branch, which is part of the precondition, cannot be implied by the postcondition for the entry block because it depends on $back_{\mathrm{tx}}$, which has been concurrently modified by the ISR and hidden away in the postcondition of the Hoare triple for the entry block. There is benign racing here.

The deep reason, however, is that the condition for the exit branch (no room in tx buffer) is not an invariant maintained by the ISR, which may increase the available room by removing data at the end of the tx queue. I call the exit branch a nonmonotonic branch, and the working branch a monotonic branch.

At the entry point, the available room in the tx queue is r_1. Shortly after the entry, the available room is queried and value r_2 is obtained. r_2 is used later in the branch condition. If $r_2 = 0$, then putch exits. Otherwise, the work is done. When it is the time to write to the buffer, the available room is r_3.

Note that from the entry point to this point, the room does not decrease. It may have been increased by the ISR. So, we have $r_1 \leq r_2 \leq r_3$.

When Hoare triple are used to describe the behavior from the entry point to exit, r_1 is used in the precondition of the entry block, r_2 is used in the branch condition, r_3 is used in specifying the wellformness of the tx buffer in the working branch.

The entry block and the working branch can be composed under the monotonic condition $r_1 > 0$. When $r_1 = 0$, it is possible $r_2 = 0$ or $r_2 > 0$, either branch can be taken. So, one cannot find a precondition at the entry point to make the exit branch be taken.

The monotonic branch does the meaningful work and can be proved to be taken under some conditions, while the nonmonotonic branch does nothing meaningful and cannot be proven to be taken under any condition. This is an idiom that programmers should follow for benign racing.

```
c80: push {r4, r5, r6, r7, lr}
c84: ldr r6, [pc, #132]
c88: ldr r3, [pc, #132]
c8c: ldrh r7, [r6]
c90: ldrh r3, [r3]
c94: add r7, r7, #1
c98: and r7, r7, #127
c9c: cmp r3, r7
ca0: mov r4, r0
ca4: mvneq r0, #0
ca8: popeq {r4, r5, r6, r7, pc}
cac: ldr r5, [pc, #100]
cb0: bl 3e8 <disableIRQ>
cb4: ldrb r3, [r5, #4]
cb8: and r3, r3, #253
cbc: strb r3, [r5, #4]
cc0: bl 3f8 <restoreIRQ>
cc4: ldr r3, [pc, #80]
cc8: ldr r2, [r3]
ccc: cmp r2, #0
cd0: ldrhne r3, [r6]
cd4: ldrne r2, [pc, #68]
cd8: andeq r2, r4, #255
cdc: moveq r1, #1
ce0: strbne r4, [r2, r3]
ce4: streq r1, [r3]
ce8: strbeq r2, [r5]
cec: strhne r7, [r6]
cf0: bl 3e8 <disableIRQ>
cf4: ldr r3, [pc, #28]
cf8: ldrb r2, [r3, #4]
cfc: orr r2, r2, #2
d00: strb r2, [r3, #4]
d04: bl 3f8 <restoreIRQ>
d08: and r0, r4, #255
d0c: pop {r4, r5, r6, r7, pc}
d10: .word 0x40000092
d14: .word 0x4000019c
d18: .word 0xe000c000
d1c: .word 0x400001a4
d20: .word 0x40000010
```

Figure 6.1: ARM assembly code for uart0Putch

```
000003e8 <disableIRQ>:
 3e8: mrs r0, CPSR
 3ec: orr r3, r0, #128
 3f0: msr CPSR_fc, r3
 3f4: bx lr
```

```
000003f8 <restoreIRQ>:
 3f8: mrs r2, CPSR
 3fc: and r0, r0, #128
 400: bic r3, r2, #128
 404: orr r3, r3, r0
 408: msr CPSR_fc, r3
 40c: mov r0, r2
 410: bx lr
```

Figure 6.2: ARM assembly code for disableIRQ and restoreIRQ

$$\left.\begin{array}{l}
\mathsf{aU}\ \mathit{divisor}\ \mathit{cushion}_{\mathrm{min}}\ \mathit{cushion}\ \mathit{strm}_{\mathrm{rx0}}\ \mathit{str}_{\mathrm{rxbuf0}}\ \mathit{str}_{\mathrm{tx}}\ \mathit{str}_{\mathrm{txout}}\ \mathit{str}_{\mathrm{txbuf}}\ \mathit{str}_{\mathrm{rx}} \\
\mathit{str}_{\mathrm{rxin}}\ \mathit{str}_{\mathrm{rxbuf}}\ \mathit{start}_{\mathrm{rx}}\ \mathit{size}_{\mathrm{rx}}\ \mathit{start}_{\mathrm{tx}}\ \mathit{size}_{\mathrm{tx}}\ \mathit{thre}\ \mathit{ie}_{\mathrm{rls}}\ \mathit{ie}_{\mathrm{rda}}\ \mathit{ie}_{\mathrm{thre}}\ \mathit{irqs}\ \mathit{df}_{\mathrm{rx}}\ f_{\mathrm{rx}} \\
\mathit{df}_{\mathrm{tx}}\ f_{\mathrm{tx}}\ \mathit{front}_{\mathrm{rx}}\ \mathit{back}_{\mathrm{rx}}\ \mathit{front}_{\mathrm{tx}}\ \mathit{back}_{\mathrm{tx}}\ \mathit{ps}\ *
\end{array}\right\} \qquad \text{(i)}$$

$$\mathsf{aBYTE\_MEMORY}\ \mathit{df}_{\mathrm{g}}\ f_{\mathrm{g}}\ * \qquad \text{(ii)}$$

$$\mathsf{cond}$$

$$\left.\begin{array}{l}
((f_{\mathrm{g}}\ \nearrow^{32}_{2}\ \mathit{front\_addr}_{\mathrm{tx}} = \mathit{front}_{\mathrm{tx}}) \wedge ((f_{\mathrm{g}}\ \nearrow^{32}_{2}\ \mathit{back\_addr}_{\mathrm{tx}} = \mathit{back}_{\mathrm{tx}}) \wedge \\
((f_{\mathrm{g}}\ \nearrow^{32}_{2}\ \mathit{front\_addr}_{\mathrm{rx}} = \mathit{front}_{\mathrm{rx}}) \wedge ((f_{\mathrm{g}}\ \nearrow^{32}_{2}\ \mathit{back\_addr}_{\mathrm{rx}} = \mathit{back}_{\mathrm{rx}}) \wedge \\
((f_{\mathrm{g}}\ \nearrow^{32}_{4}\ \mathit{tx\_running\_addr} = \mathit{tx\_running})\ \wedge
\end{array}\right\} \qquad \text{(iii)}$$

$$\left.\begin{array}{l}
(\overleftarrow{\mathit{front\_addr}_{\mathrm{rx}}}^{2} \cup \overleftarrow{\mathit{back\_addr}_{\mathrm{rx}}}^{2} \cup \overleftarrow{\mathit{front\_addr}_{\mathrm{tx}}}^{2} \cup \overleftarrow{\mathit{back\_addr}_{\mathrm{tx}}}^{2} \cup \\
\overleftarrow{\mathit{tx\_running\_addr}}^{4}) \subseteq \mathit{df}_{\mathrm{g}} \wedge
\end{array}\right\} \qquad \text{(iv)}$$

$$\mathsf{circ\_space}\ \mathit{size}_{\mathrm{tx}}\ \mathit{front}_{\mathrm{tx}}\ \mathit{back}_{\mathrm{tx}} \geq_{+} \mathit{space}_{\mathrm{tx0}} \wedge \qquad \text{(v)}$$

$$((\mathit{tx\_running} = 0) \Rightarrow (\mathit{front}_{\mathrm{tx}} = \mathit{back}_{\mathrm{tx}}) \wedge \mathit{thre})) \qquad \text{(vi)}$$

Figure 6.3: Formula used to assert the ISR in interrupt mode

$$\left.\begin{array}{l}\mathsf{aU}\ \mathit{divisor}\ \mathit{cushion}_{\mathrm{min}}\ \mathit{cushion}\ \mathit{strm}_{\mathrm{rx0}}\ \mathit{str}_{\mathrm{rxbuf0}}\ \mathit{str}_{\mathrm{tx}}\ \mathit{str}_{\mathrm{txout}}\ \mathit{str}_{\mathrm{txbuf}}\\ \mathit{str}_{\mathrm{rx}}\ \mathit{str}_{\mathrm{rxin}}\ \mathit{str}_{\mathrm{rxbuf}}\ \mathit{start}_{\mathrm{rx}}\ \mathit{size}_{\mathrm{rx}}\ \mathit{start}_{\mathrm{tx}}\ \mathit{size}_{\mathrm{tx}}\ \mathit{thre}\ \mathit{ie}_{\mathrm{rls}}\ \mathit{ie}_{\mathrm{rda}}\ \mathit{ie}_{\mathrm{thre}}\ \mathit{irqs}\\ \mathit{df}_{\mathrm{rx}}\ f_{\mathrm{rx}}\ \mathit{df}_{\mathrm{tx}}\ f_{\mathrm{tx}}\ \mathit{front}_{\mathrm{rx}}\ \mathit{back}_{\mathrm{rx}}\ \mathit{front}_{\mathrm{tx}}\ \mathit{back}_{\mathrm{tx}}\ \mathit{ps}\ *\end{array}\right\}\qquad\text{(i)}$$

$$\mathsf{aBYTE\_MEMORY}\ \mathit{df}_{\mathrm{g}}\ f_{\mathrm{g}}\ * \qquad \text{(ii)}$$

$$\mathsf{cond}$$

$$\left.\begin{array}{l}((f_{\mathrm{g}}\ \nearrow^{32}_{2}\ \mathit{front\_addr}_{\mathrm{tx}} = \mathit{front}_{\mathrm{tx}}) \wedge ((f_{\mathrm{g}}\ \nearrow^{32}_{2}\ \mathit{back\_addr}_{\mathrm{tx}} = \mathit{back}_{\mathrm{tx}}) \wedge\\ ((f_{\mathrm{g}}\ \nearrow^{32}_{2}\ \mathit{front\_addr}_{\mathrm{rx}} = \mathit{front}_{\mathrm{rx}}) \wedge ((f_{\mathrm{g}}\ \nearrow^{32}_{2}\ \mathit{back\_addr}_{\mathrm{rx}} = \mathit{back}_{\mathrm{rx}}) \wedge\\ ((f_{\mathrm{g}}\ \nearrow^{32}_{4}\ \mathit{tx\_running\_addr} = \mathit{tx\_running})\ \wedge\end{array}\right\}\qquad\text{(iii)}$$

$$\left.\begin{array}{l}(\overleftarrow{\mathit{front\_addr}_{\mathrm{rx}}}^{2} \cup \overleftarrow{\mathit{back\_addr}_{\mathrm{rx}}}^{2} \cup \overleftarrow{\mathit{front\_addr}_{\mathrm{tx}}}^{2} \cup \overleftarrow{\mathit{back\_addr}_{\mathrm{tx}}}^{2} \cup\\ \overleftarrow{\mathit{tx\_running\_addr}}^{4}) \subseteq \mathit{df}_{\mathrm{g}} \wedge\end{array}\right\}\qquad\text{(iv)}$$

$$\mathsf{circ\_space}\ \mathit{size}_{\mathrm{tx}}\ \mathit{front}_{\mathrm{tx}}\ \mathit{back}_{\mathrm{tx}} \geq_{+} \mathit{space}_{\mathrm{tx0}} \wedge \qquad \text{(v)}$$

$$((\mathit{tx\_running} = 0) \Rightarrow (\mathit{front}_{\mathrm{tx}} = \mathit{back}_{\mathrm{tx}}) \wedge \mathit{thre}) \wedge \qquad \text{(vi)}$$

$$\neg\mathsf{MEM}\ \mathsf{IRQ}_{\mathrm{THRE}}\ \mathit{irqs}) \qquad \text{(vii)}$$

Figure 6.4: Formula used to assert the ISR in THRE-atomic mode

$$\left.\begin{array}{l}\mathsf{aBYTE\_MEMORY}\ df_{\mathrm{rx}}\ f_{\mathrm{rx}} * \mathsf{aBYTE\_MEMORY}\ df_{\mathrm{tx}}\ f_{\mathrm{tx}} * \\ \mathsf{aBYTE\_MEMORY}\ df_{\mathrm{g}}\ f_{\mathrm{g}} *\ \mathsf{aP}\ ps * \end{array}\right\} \qquad \text{(i)}$$

$$\mathsf{cond}$$

$$\left.\begin{array}{l}((\mathsf{get\_divisor}\ ps = divisor) \wedge (\mathsf{uart\_pending\_irqs}\ ps = irqs) \wedge \\ (ps.\mathsf{IER}_{\mathrm{RLS}} \iff ie_{\mathrm{rls}}) \wedge (ps.\mathsf{IER}_{\mathrm{RDA}} \iff ie_{\mathrm{rda}}) \wedge \\ (ps.\mathsf{IER}_{\mathrm{THRE}} \iff ie_{\mathrm{thre}}) \wedge (ps.\mathsf{LSR}_{\mathrm{THRE}} \iff thre) \wedge \end{array}\right\} \qquad \text{(ii)}$$

$$\left.\begin{array}{l}(f_{\mathrm{g}} \nearrow^{32}_{2}\ front\_addr_{\mathrm{tx}} = front_{\mathrm{tx}}) \wedge (f_{\mathrm{g}} \nearrow^{32}_{2}\ back\_addr_{\mathrm{tx}} = back_{\mathrm{tx}}) \wedge \\ (f_{\mathrm{g}} \nearrow^{32}_{2}\ front\_addr_{\mathrm{rx}} = front_{\mathrm{rx}}) \wedge (f_{\mathrm{g}} \nearrow^{32}_{2}\ back\_addr_{\mathrm{rx}} = back_{\mathrm{rx}}) \wedge \\ (f_{\mathrm{g}} \nearrow^{32}_{4}\ tx\_running\_addr = tx\_running)\ \wedge \end{array}\right\} \qquad \text{(iii)}$$

$$\left.\begin{array}{l}(\overleftarrow{front\_addr_{\mathrm{rx}}}^{2} \cup \overleftarrow{back\_addr_{\mathrm{rx}}}^{2} \cup \overleftarrow{front\_addr_{\mathrm{tx}}}^{2} \cup \overleftarrow{back\_addr_{\mathrm{tx}}}^{2} \cup \\ \overleftarrow{tx\_running\_addr}^{4}) \subseteq df_{\mathrm{g}} \wedge \end{array}\right\} \qquad \text{(iv)}$$

$$\mathsf{uart0.wellform}\ ps \wedge \neg ps.\mathsf{LCR}_{\mathrm{DLAB}} \wedge \qquad \text{(v)}$$

$$cushion + divisor_{\mathrm{min}} < divisor \wedge \qquad \text{(vi)}$$

$$(\neg \mathsf{NULL}\ (\mathsf{uart\_pending\_irqs}\ ps) \Rightarrow ps.\mathsf{clock} + cushion + cushion_{\mathrm{min}} < divisor) \wedge \qquad \text{(vii)}$$

$$\left.\begin{array}{l}\mathsf{sentString}\ str_{\mathrm{txout}}\ ps \wedge (\mathsf{REVERSE}\ str_{\mathrm{txbuf}} \mathbin{+\!+} str_{\mathrm{txout}} = str_{\mathrm{tx}}) \wedge \\ \mathsf{circBuf}\ start_{\mathrm{tx}}\ size_{\mathrm{tx}}\ df_{\mathrm{tx}}\ f_{\mathrm{tx}}\ front_{\mathrm{tx}}\ back_{\mathrm{tx}}\ str_{\mathrm{txbuf}} \wedge \\ \mathsf{circ\_space}\ size_{\mathrm{tx}}\ front_{\mathrm{tx}}\ back_{\mathrm{tx}} \geq_{+} space_{\mathrm{tx0}} \wedge \\ ((tx\_running = 0) \Rightarrow (front_{\mathrm{tx}} = back_{\mathrm{tx}}) \wedge thre) \end{array}\right\} \qquad \text{(viii)}$$

$$\left.\begin{array}{l}\mathsf{inputString}\ str_{\mathrm{rxin}}\ ps \wedge (str_{\mathrm{rxbuf}} \mathbin{+\!+} str_{\mathrm{rxin}} = str_{\mathrm{rx}}) \wedge \\ \mathsf{circBuf}\ start_{\mathrm{rx}}\ size_{\mathrm{rx}}\ df_{\mathrm{rx}}\ f_{\mathrm{rx}}\ front_{\mathrm{rx}}\ back_{\mathrm{rx}}\ str_{\mathrm{rxbuf}} \wedge \\ (ps.\mathsf{LSR}_{\mathrm{RDR}} \Rightarrow \neg \mathsf{NULL}\ str_{\mathrm{rxin}}) \wedge \mathsf{ishifted}\ strm_{\mathrm{rx0}}\ str_{\mathrm{rxbuf0}}\ ps\ str_{\mathrm{rxbuf}}) \end{array}\right\} \qquad \text{(ix)}$$

Figure 6.5: Formula used to assert uart0Putch in interrupt mode and atomic mode

$$\left.\begin{array}{l}\mathsf{aBYTE\_MEMORY}\ df_{\mathrm{rx}}\ f_{\mathrm{rx}} * \mathsf{aBYTE\_MEMORY}\ df_{\mathrm{tx}}\ f_{\mathrm{tx}} * \\ \mathsf{aBYTE\_MEMORY}\ df_{\mathrm{g}}\ f_{\mathrm{g}} * \ \mathsf{aP}\ ps * \end{array}\right\} \qquad \text{(i)}$$

$$\mathsf{cond}$$

$$\left.\begin{array}{l}((\mathsf{get\_divisor}\ ps = divisor) \wedge (\mathsf{uart\_pending\_irqs}\ ps = irqs) \wedge \\ (ps.\mathsf{IER}_{\mathrm{RLS}} \iff ie_{\mathrm{rls}}) \wedge (ps.\mathsf{IER}_{\mathrm{RDA}} \iff ie_{\mathrm{rda}}) \wedge \\ (ps.\mathsf{IER}_{\mathrm{THRE}} \iff ie_{\mathrm{thre}}) \wedge (ps.\mathsf{LSR}_{\mathrm{THRE}} \iff thre) \wedge \end{array}\right\} \qquad \text{(ii)}$$

$$\left.\begin{array}{l}(f_{\mathrm{g}} \nearrow^{32}_{2} front\_addr_{\mathrm{tx}} = front_{\mathrm{tx}}) \wedge (f_{\mathrm{g}} \nearrow^{32}_{2} back\_addr_{\mathrm{tx}} = back_{\mathrm{tx}}) \wedge \\ (f_{\mathrm{g}} \nearrow^{32}_{2} front\_addr_{\mathrm{rx}} = front_{\mathrm{rx}}) \wedge (f_{\mathrm{g}} \nearrow^{32}_{2} back\_addr_{\mathrm{rx}} = back_{\mathrm{rx}}) \wedge \\ (f_{\mathrm{g}} \nearrow^{32}_{4} tx\_running\_addr = tx\_running) \wedge \end{array}\right\} \qquad \text{(iii)}$$

$$\left.\begin{array}{l}(\overleftarrow{front\_addr_{\mathrm{rx}}}^{2} \cup \overleftarrow{back\_addr_{\mathrm{rx}}}^{2} \cup \overleftarrow{front\_addr_{\mathrm{tx}}}^{2} \cup \overleftarrow{back\_addr_{\mathrm{tx}}}^{2} \cup \\ \overleftarrow{tx\_running\_addr}^{4}) \subseteq df_{\mathrm{g}} \wedge \end{array}\right\} \qquad \text{(iv)}$$

$$\mathsf{uart0.wellform}\ ps \wedge \neg ps.\mathsf{LCR}_{\mathrm{DLAB}} \wedge \qquad \text{(v)}$$

$$cushion + divisor_{\mathrm{min}} < divisor \wedge \qquad \text{(vi)}$$

$$(\neg\mathsf{NULL}\ (\mathsf{uart\_pending\_irqs}\ ps) \Rightarrow ps.\mathsf{clock} + cushion + cushion_{\mathrm{min}} < divisor) \wedge \qquad \text{(vii)}$$

$$\left.\begin{array}{l}\mathsf{sentString}\ str_{\mathrm{txout}}\ ps \wedge (\mathsf{REVERSE}\ str_{\mathrm{txbuf}}\mathsf{++}str_{\mathrm{txout}} = str_{\mathrm{tx}}) \wedge \\ \mathsf{circBuf}\ start_{\mathrm{tx}}\ size_{\mathrm{tx}}\ df_{\mathrm{tx}}\ f_{\mathrm{tx}}\ front_{\mathrm{tx}}\ back_{\mathrm{tx}}\ str_{\mathrm{txbuf}} \wedge \\ \mathsf{circ\_space}\ size_{\mathrm{tx}}\ front_{\mathrm{tx}}\ back_{\mathrm{tx}} \geq_{+} space_{\mathrm{tx0}} \wedge \\ ((tx\_running = 0) \Rightarrow (front_{\mathrm{tx}} = back_{\mathrm{tx}}) \wedge thre) \end{array}\right\} \qquad \text{(viii)}$$

$$\left.\begin{array}{l}\mathsf{inputString}\ str_{\mathrm{rxin}}\ ps \wedge (str_{\mathrm{rxbuf}}\mathsf{++}str_{\mathrm{rxin}} = str_{\mathrm{rx}}) \wedge \\ \mathsf{circBuf}\ start_{\mathrm{rx}}\ size_{\mathrm{rx}}\ df_{\mathrm{rx}}\ f_{\mathrm{rx}}\ front_{\mathrm{rx}}\ back_{\mathrm{rx}}\ str_{\mathrm{rxbuf}} \wedge \\ (ps.\mathsf{LSR}_{\mathrm{RDR}} \Rightarrow \neg\mathsf{NULL}\ str_{\mathrm{rxin}}) \wedge \mathsf{ishifted}\ strm_{\mathrm{rx0}}\ str_{\mathrm{rxbuf0}}\ ps\ str_{\mathrm{rxbuf}} \wedge \end{array}\right\} \qquad \text{(ix)}$$

$$\neg\mathsf{MEM}\ \mathsf{IRQ}_{\mathrm{THRE}}\ irqs) \qquad \text{(x)}$$

Figure 6.6: Formula used to assert uart0Putch in THRE-atomic mode

Table 6.1: High-level view of low-level system resources

Low-level	High-level
ps	$divisor$, $irqs$,$thre$, ie_{rls}, ie_{rda}, ie_{thre},str_{rxin}, str_{rx},str_{txout}, str_{tx}
(f_{rx},df_{rx})	str_{rx}, str_{rxbuf}
(f_{tx},df_{tx})	str_{tx}, str_{txbuf}
(f_{g},df_{g})	$front_{\mathrm{rx}}$, $back_{\mathrm{rx}}$,$front_{\mathrm{tx}}$, $back_{\mathrm{tx}}$, $tx\_running$

Table 6.2: Resource conflicts between uart0Putch and the ISR

uart0Putch access	ISR access	Resource
w	r	str_{txout}, str_{txbuf},$thre$, ie_{thre},$back_{\mathrm{tx}}$,$tx\_running$,$irqs$,
r	w	$front_{\mathrm{tx}}$, str_{txout}, str_{txbuf},$thre$, $tx\_running$

$$\mathsf{aU}_1\ \mathit{divisor}\ \mathit{cushion}_{\mathrm{min}}\ \mathit{cushion}\ \mathit{ie}_{\mathrm{rls}}\ \mathit{ie}_{\mathrm{rda}}\ \mathit{ie}_{\mathrm{thre}}\ \mathit{irqs}\ tx_r unning_a ddr\ \mathit{df}_{\mathrm{g}}\ \mathit{start}_{\mathrm{tx}}\ \mathit{size}_{\mathrm{tx}}\ \mathit{front\_addr}_{\mathrm{tx}}\ \mathit{back\_addr}_{\mathrm{tx}}\ \mathit{df}_{\mathrm{tx}}\ f_{\mathrm{tx}}\ \mathit{str}_{\mathrm{tx}}\ \mathit{spaceoe}\ \mathit{back}_{\mathrm{tx}}\ \mathit{start}_{\mathrm{rx}}\ \mathit{size}_{\mathrm{rx}}\ \mathit{front\_addr}_{\mathrm{rx}}\ \mathit{back\_addr}_{\mathrm{rx}}\ \mathit{df}_{\mathrm{rx}}\ \mathit{strm}_{\mathrm{rx0}}\ \mathit{str}_{\mathrm{rxbuf0}}\ \mathit{front}_{\mathrm{rx}}\ (\mathit{str}_{\mathrm{txout}},\ \mathit{str}_{\mathrm{txbuf}},\ \mathit{str}_{\mathrm{rx}},\ \mathit{str}_{\mathrm{rxin}},\ \mathit{str}_{\mathrm{rxbuf}},\ f_{\mathrm{rx}},\ f_{\mathrm{g}},\ \mathit{front}_{\mathrm{tx}},\ \mathit{back}_{\mathrm{rx}},\ \mathit{tx\_running},\ \mathit{thre},\ \mathit{ps}) \tag{i}$$

$$\mathsf{aU}_2\ \mathit{divisor}\ \mathit{cushion}_{\mathrm{min}}\ \mathit{cushion}\ \mathit{thre}\ \mathit{ie}_{\mathrm{rls}}\ \mathit{ie}_{\mathrm{rda}}\ \mathit{ie}_{\mathrm{thre}}\ \mathit{irqs}\ tx_r unning_a ddr\ \mathit{df}_{\mathrm{g}}\ \mathit{start}_{\mathrm{tx}}\ \mathit{size}_{\mathrm{tx}}\ \mathit{front\_addr}_{\mathrm{tx}}\ \mathit{back\_addr}_{\mathrm{tx}}\ \mathit{df}_{\mathrm{tx}}\ \mathit{str}_{\mathrm{tx}}\ \mathit{spaceoe}\ \mathit{front}_{\mathrm{tx}}\ \mathit{back}_{\mathrm{tx}}\ \mathit{str}_{\mathrm{txout}}\ \mathit{str}_{\mathrm{txbuf}}\ f_{\mathrm{tx}}\ tx_r unning\ \mathit{start}_{\mathrm{rx}}\ \mathit{size}_{\mathrm{rx}}\ \mathit{front\_addr}_{\mathrm{rx}}\ \mathit{back\_addr}_{\mathrm{rx}}\ \mathit{df}_{\mathrm{rx}}\ \mathit{strm}_{\mathrm{rx0}}\ \mathit{str}_{\mathrm{rxbuf0}}\ \mathit{front}_{\mathrm{rx}}\ (\mathit{str}_{\mathrm{rx}},\ \mathit{str}_{\mathrm{rxin}},\ \mathit{str}_{\mathrm{rxbuf}},\ f_{\mathrm{rx}},\ f_{\mathrm{g}},\ \mathit{back}_{\mathrm{rx}},\ \mathit{ps}) \tag{ii}$$

$$\mathsf{aU}_3\ \mathit{divisor}\ \mathit{cushion}_{\mathrm{min}}\ \mathit{cushion}\ \mathit{ie}_{\mathrm{rls}}\ \mathit{ie}_{\mathrm{rda}}\ \mathit{ie}_{\mathrm{thre}}\ \mathit{irqs}\ tx_r unning_a ddr\ \mathit{df}_{\mathrm{g}}\ \mathit{start}_{\mathrm{tx}}\ \mathit{size}_{\mathrm{tx}}\ \mathit{front\_addr}_{\mathrm{tx}}\ \mathit{back\_addr}_{\mathrm{tx}}\ \mathit{df}_{\mathrm{tx}}\ f_{\mathrm{tx}}\ \mathit{str}_{\mathrm{tx}}\ \mathit{spaceoe}\ \mathit{back}_{\mathrm{tx}}\ \mathit{start}_{\mathrm{rx}}y\ \mathit{size}_{\mathrm{rx}}\ \mathit{front\_addr}_{\mathrm{rx}}\ \mathit{back\_addr}_{\mathrm{rx}}\ \mathit{df}_{\mathrm{rx}}\ \mathit{strm}_{\mathrm{rx0}}\ \mathit{str}_{\mathrm{rxbuf0}}\ \mathit{front}_{\mathrm{rx}}\ (\mathit{str}_{\mathrm{txout}},\ \mathit{str}_{\mathrm{txbuf}},\ \mathit{str}_{\mathrm{rx}},\ \mathit{str}_{\mathrm{rxin}},\ \mathit{str}_{\mathrm{rxbuf}},\ f_{\mathrm{rx}},\ f_{\mathrm{g}},\ \mathit{front}_{\mathrm{tx}},\ \mathit{back}_{\mathrm{rx}},\ tx_r unning,\ \mathit{thre},\ \mathit{ps}) \tag{iii}$$

Figure 6.7: Abstraction of the assertion formula for the serial port ISR in different modes

$$\mathsf{aU}_4\ \mathit{divisor}\ \mathit{divisor}_{\rm min}\ \mathit{cushion}_{\rm min}\ \mathit{cushion}\ \mathit{ie}_{\rm rls}\ \mathit{ie}_{\rm rda}\ \mathit{ie}_{\rm thre}\ \mathit{df}_{\rm g}\ \mathit{df}_{\rm rx}\ \mathit{df}_{\rm tx}\ f_{\rm tx}\ \mathit{tx\_running\_addr}\ \mathit{front\_addr}_{\rm tx}\ \mathit{back\_addr}_{\rm tx}\ \mathit{front\_addr}_{\rm rx}\ \mathit{back\_addr}_{\rm rx}\ \mathit{start}_{\rm tx}\ \mathit{size}_{\rm tx}\ \mathit{start}_{\rm rx}\ \mathit{size}_{\rm rx}\ \mathit{str}_{\rm tx}\ \mathit{spaceoe}\ \mathit{back}_{\rm tx}\ \mathit{strm}_{\rm rx0}\ \mathit{str}_{\rm rxbuf0}\ \mathit{front}_{\rm rx}\ (\mathit{str}_{\rm txout},\ \mathit{str}_{\rm txbuf},\ \mathit{str}_{\rm rx},\ \mathit{str}_{\rm rxin},\ \mathit{str}_{\rm rxbuf},\ \mathit{irqs},\ f_{\rm rx},\ f_{\rm g},\ \mathit{front}_{\rm tx},\ \mathit{back}_{\rm rx},\ \mathit{tx\_running},\ \mathit{thre},\ \mathit{ps}) \qquad \text{(i)}$$

$$\mathsf{aU}_5\ \mathit{divisor}\ \mathit{divisor}_{\rm min}\ \mathit{cushion}_{\rm min}\ \mathit{cushion}\ \mathit{ie}_{\rm rls}\ \mathit{ie}_{\rm rda}\ \mathit{ie}_{\rm thre}\ \mathit{df}_{\rm g}\ \mathit{df}_{\rm rx}\ \mathit{df}_{\rm tx}\ f_{\rm tx}\ \mathit{tx\_running}\ \mathit{tx\_running\_addr}\ \mathit{front\_addr}_{\rm tx}\ \mathit{back\_addr}_{\rm tx}\ \mathit{front\_addr}_{\rm rx}\ \mathit{back\_addr}_{\rm rx}\ \mathit{start}_{\rm tx}\ \mathit{size}_{\rm tx}\ \mathit{start}_{\rm rx}\ \mathit{size}_{\rm rx}\ \mathit{str}_{\rm tx}\ \mathit{str}_{\rm txout}\ \mathit{str}_{\rm txbuf}\ \mathit{front}_{\rm tx}\ \mathit{back}_{\rm tx}\ \mathit{spaceoe}\ \mathit{strm}_{\rm rx0}\ \mathit{str}_{\rm rxbuf0}\ \mathit{front}_{\rm rx}\ (\mathit{str}_{\rm rx},\ \mathit{str}_{\rm rxin},\ \mathit{str}_{\rm rxbuf},\ \mathit{irqs},\ f_{\rm rx},\ f_{\rm g},\ \mathit{back}_{\rm i},\ \mathit{thre},\ \mathit{ps}) \qquad \text{(ii)}$$

$$\mathsf{aU}_6\ \mathit{divisor}\ \mathit{divisor}_{\rm min}\ \mathit{cushion}_{\rm min}\ \mathit{cushion}\ \mathit{ie}_{\rm rls}\ \mathit{ie}_{\rm rda}\ \mathit{ie}_{\rm thre}\ \mathit{df}_{\rm g}\ \mathit{df}_{\rm rx}\ \mathit{df}_{\rm tx}\ f_{\rm tx}\ \mathit{thre}\ \mathit{tx\_running}\ \mathit{tx\_running\_addr}\ \mathit{front\_addr}_{\rm tx}\ \mathit{back\_addr}_{\rm tx}\ \mathit{front\_addr}_{\rm rx}\ \mathit{back\_addr}_{\rm rx}\ \mathit{start}_{\rm tx}\ \mathit{size}_{\rm tx}\mathit{start}_{\rm rx}\ \mathit{size}_{\rm rx}\ \mathit{str}_{\rm tx}\ \mathit{str}_{\rm txout}\ \mathit{str}_{\rm txbuf}\ \mathit{front}_{\rm tx}\ \mathit{back}_{\rm tx}\ \mathit{spaceoe}\ \mathit{strm}_{\rm rx0}\ \mathit{str}_{\rm rxbuf0}\ \mathit{front}_{\rm rx}\ (\mathit{str}_{\rm rx},\ \mathit{str}_{\rm rxin},\ \mathit{str}_{\rm rxbuf},\ \mathit{irqs},\ f_{\rm rx},\ f_{\rm g},\ \mathit{back}_{\rm rx},\ \mathit{ps}) \qquad \text{(iii)}$$

$$\mathsf{aU}_7\ \mathit{divisor}\ \mathit{divisor}_{\rm min}\ \mathit{cushion}_{\rm min}\ \mathit{cushion}\ \mathit{ie}_{\rm rls}\ \mathit{ie}_{\rm rda}\ \mathit{ie}_{\rm thre}\ \mathit{df}_{\rm g}\ \mathit{df}_{\rm rx}\ \mathit{df}_{\rm tx}\ f_{\rm tx}\ \mathit{tx\_running\_addr}\ \mathit{front\_addr}_{\rm tx}\ \mathit{back\_addr}_{\rm tx}\ \mathit{front\_addr}_{\rm rx}\ \mathit{back\_addr}_{\rm rx}\ \mathit{start}_{\rm tx}\ \mathit{size}_{\rm tx}\ \mathit{start}_{\rm rx}\ \mathit{size}_{\rm rx}\ \mathit{str}_{\rm tx}\ \mathit{spaceoe}\ \mathit{back}_{\rm tx}\ \mathit{strm}_{\rm rx0}\ \mathit{str}_{\rm rxbuf0}\ \mathit{front}_{\rm rx}\ (\mathit{str}_{\rm txout},\ \mathit{str}_{\rm txbuf},\ \mathit{str}_{\rm rx},\ \mathit{str}_{\rm rxin},\ \mathit{str}_{\rm rxbuf},\ \mathit{irqs},\ f_{\rm rx},\ f_{\rm g},\ \mathit{front}_{\rm tx},\ \mathit{back}_{\rm rx},\ \mathit{tx\_running},\ \mathit{thre},\ \mathit{ps}) \qquad \text{(iv)}$$

$$\mathsf{aU}_8\ \mathit{divisor}\ \mathit{divisor}_{\rm min}\ \mathit{cushion}_{\rm min}\ \mathit{cushion}\ \mathit{ie}_{\rm rls}\ \mathit{ie}_{\rm rda}\ \mathit{ie}_{\rm thre}\ \mathit{df}_{\rm g}\ \mathit{df}_{\rm rx}\ \mathit{df}_{\rm tx}\ \mathit{tx\_running\_addr}\ \mathit{front\_addr}_{\rm tx}\ \mathit{back\_addr}_{\rm tx}\ \mathit{front\_addr}_{\rm rx}\ \mathit{back\_addr}_{\rm rx}\ \mathit{start}_{\rm tx}\ \mathit{size}_{\rm tx}\ \mathit{start}_{\rm rx}\ \mathit{size}_{\rm rx}\ \mathit{str}_{\rm tx}\ \mathit{spaceoe}\ \mathit{strm}_{\rm rx0}\ \mathit{str}_{\rm rxbuf0}\ \mathit{front}_{\rm rx}\ (\mathit{str}_{\rm txout},\ \mathit{str}_{\rm txbuf},\ \mathit{str}_{\rm rx},\ \mathit{str}_{\rm rxin},\ \mathit{str}_{\rm rxbuf},\ \mathit{irqs},\ f_{\rm rx},\ f_{\rm tx},\ f_{\rm g},\ \mathit{front}_{\rm tx},\ \mathit{back}_{\rm tx},\ \mathit{back}_{\rm rx},\ \mathit{tx\_running},\ \mathit{thre},\ \mathit{ps}) \qquad \text{(v)}$$

Figure 6.8: Abstraction of the assertion formula for u0Putch in different modes

$\vdash$ SPEC LPC_IRQ_MODEL

(aCPSR $cpsr$ $*$ aMD 18 $*$ aPC 316 $*$ aPSR ips $ipsr$ $*$ aR LRirq $lrirq$ $*$
aR R_0 r_0 $*$ aR R_1 r_1 $*$ aR R_{10} r_{10} $*$ aR R_{11} r_{11} $*$ aR R_{12} r_{12} $*$
aR R_2 r_2 $*$ aR R_3 r_3 $*$ aR R_4 r_4 $*$ aR R_5 r_5 $*$ aR R_6 r_6 $*$ aR R_7 r_7 $*$
aR R_8 r_8 $*$ aR R_9 r_9 $*$ aS1 psrA $psra$ $*$ aS1 psrC $psrc$ $*$ aS1 psrI T $*$
aS1 psrN $psrn$ $*$ aS1 psrQ $psrq$ $*$ aS1 psrV $psrv$ $*$ aS1 psrZ $psrz$ $*$
cond
$(ipsr.\mathsf{M} \in \{16, 17, 18, 19, 23, 27, 31\} \wedge (3 \mathbin{\&\&} lrirq - 4 = 0) \wedge$
$(3 \mathbin{\&\&} spirq - 60 = 0) \wedge$
$(3 \mathbin{\&\&} spirq - 56 = 0) \wedge ipsr.\mathsf{F} \wedge (ipsr.\mathsf{IT} = 0) \wedge$
$\neg ipsr.\mathsf{E} \wedge \neg ipsr.\mathsf{J} \wedge \neg ipsr.\mathsf{T})$ $*$ aPPSTACK irq $spirq$ [] 15 $*$
aU_1 $divisor$ 85 $cushion$ ie_{rls} ie_{rda} ie_{thre} $irqs$ $\underline{tx\_running\_addr}$ df
$\underline{start_{\mathrm{tx}}}$ $\underline{size_{\mathrm{tx}}}$ $\underline{front\_addr_{\mathrm{tx}}}$ $\underline{back\_addr_{\mathrm{tx}}}$ df_{tx} f_{tx} str_{tx} $spaceoe$
$back_{\mathrm{tx}}\underline{start_{\mathrm{rx}}}$ $\underline{size_{\mathrm{rx}}}$ $\underline{front\_addr_{\mathrm{rx}}}$ $\underline{back\_addr_{\mathrm{rx}}}$ df_{rx} $strm_{\mathrm{rx0}}$ str_{rxbuf0}
$front_{\mathrm{rx}}$
$(str_{\mathrm{txout}}, str_{\mathrm{txbuf}}, str_{\mathrm{rx}}, str_{\mathrm{rxin}}, str_{\mathrm{rxbuf}}, f_{\mathrm{rx}}, f, front_{\mathrm{tx}}, back_{\mathrm{rx}}, tx\_running, thre,$
$ps))$
isr_code $\cup$ driver_code
(aCPSR ($cpsr with$ ge := $ipsr$.GE) $*$ aMD$ipsr$.M $*$
aPC $(lrirq - 4)$ $*$ aPSR ips $ipsr$ $*$
aR $LRirq$ $(lrirq - 4)$ $*$ aR R_0 r_0 $*$ aR R_1 r_1 $*$
aR R_{10} r_{10} $*$ aR R_{11} r_{11} $*$ aR R_{12} r_{12} $*$ aR $\mathsf{R}_2 r_2$ $*$ aR $\mathsf{R}_3 r_3$ $*$
aR R_4 r_4 $*$ aR R_5 r_5 $*$ aR R_6 r_6 $*$ aRR_7 r_7 $*$ aR R_8 r_8 $*$ aR R_9 r_9 $*$
aS1 psrA $ipsr$.A $*$ aS1 psrC $ipsr$.C $*$ aS1 psrI $ipsr$.I $*$
aS1 psrN $ipsr$.N $*$ aS1 psrQ $ipsr$.Q $*$ aS1 psrV $ipsr$.V $*$
aS1 psrZ $ipsr$.Z $*$
$\neg$aU_1 $divisor$ 0 $cushion$ ie_{rls} ie_{rda} ie_{thre} [] $\underline{tx\_running\_addr}$ df $\underline{start_{\mathrm{tx}}}$
$\underline{size_{\mathrm{tx}}}$ $\underline{front\_addr_{\mathrm{tx}}}$ $\underline{back\_addr_{\mathrm{tx}}}$ df_{tx} f_{tx} str_{tx} $spaceoe$ $back_{\mathrm{tx}}$
$\underline{start_{\mathrm{rx}}}$ $\underline{size_{\mathrm{rx}}}$ $\underline{front\_addr_{\mathrm{rx}}}$ $\underline{back\_addr_{\mathrm{rx}}}$ df_{rx} $strm_{\mathrm{rx0}}$ str_{rxbuf0}
$front_{\mathrm{rx}}$ $*$ aPPSTACK irq $spirq$ [] 15)

Figure 6.9: Hoare triple for the ISR in interrupt mode

$\vdash$ SPEC LPC_IRQ_MODEL
(aCPSR $cpsr$ $*$ aMD 18 $*$ aPC 316 $*$ aPSR $ips\ ipsr$ $*$ aR LRirq $lrirq$ $*$
aR R_0 r_0 $*$ aR R_1 r_1 $*$ aR R_{10} r_{10} $*$ aR R_{11} r_{11} $*$ aR R_{12} r_{12} $*$
aR R_2 r_2 $*$ aR R_3 r_3 $*$ aR R_4 r_4 $*$ aR R_5 r_5 $*$ aR R_6 r_6 $*$ aR R_7 r_7 $*$
aR R_8 r_8 $*$ aR R_9 r_9 $*$ aS1 psrA $psra$ $*$ aS1 psrC $psrc$ $*$ aS1 psrI T $*$
aS1 psrN $psrn$ $*$ aS1 psrQ $psrq$ $*$ aS1 psrV $psrv$ $*$ aS1 psrZ $psrz$ $*$
cond
$(ipsr.\mathsf{M} \in \{16, 17, 18, 19, 23, 27, 31\} \wedge (3\ \&\&\ lrirq - 4 = 0) \wedge$
$(3\ \&\&\ spirq - 60 = 0) \wedge$
$(3\ \&\&\ spirq - 56 = 0) \wedge ipsr.\mathsf{F} \wedge (ipsr.\mathsf{IT} = 0) \wedge$
$\neg ipsr.\mathsf{E} \wedge \neg ipsr.\mathsf{J} \wedge \neg ipsr.\mathsf{T})$ $*$ aPPSTACK $irq\ spirq$ $[]$ 15 $*$
aU$_2$ $divisor\ 85\ cushion\ ie_{\mathrm{rls}}\ ie_{\mathrm{rda}}$ F $irqs\ \underline{tx\_running\_addr}\ df$
$\underline{start_{\mathrm{tx}} size_{\mathrm{tx}}}\ \underline{front\_addr_{\mathrm{tx}}}\ \underline{back\_addr_{\mathrm{tx}}}\ df_{\mathrm{tx}}\ str_{\mathrm{tx}}\ spaceoe\ front_{\mathrm{tx}}$
$back_{\mathrm{tx}}\ str_{\mathrm{txout}}\ str_{\mathrm{txbuf}}\ f_{\mathrm{tx}}\ tx\_running\ \underline{start_{\mathrm{rx}}}\ \underline{size_{\mathrm{rx}}}\ \underline{front\_addr_{\mathrm{rx}}}\ \underline{back\_addr_{\mathrm{rx}}}$
$df_{\mathrm{rx}}\ strm_{\mathrm{rx0}}\ str_{\mathrm{rxbuf0}}\ front_{\mathrm{rx}}$
$(str_{\mathrm{rx}}, str_{\mathrm{rxin}}, str_{\mathrm{rxbuf}}, f_{\mathrm{rx}}, f, back_{\mathrm{rx}},\ ps))$
isr_code $\cup$ driver_code
(aCPSR $(cpsr with$ ge $:=$ $ipsr.$GE) $*$ aMD$ipsr.$M $*$
aPC $(lrirq - 4)$ $*$ aPSR $ips\ ipsr$ $*$
aR $LRirq$ $(lrirq - 4)$ $*$ aR R_0 r_0 $*$ aR R_1 r_1 $*$
aR R_{10} r_{10} $*$ aR R_{11} r_{11} $*$ aR R_{12} r_{12} $*$ aR $\mathsf{R}_2 r_2$ $*$ aR $\mathsf{R}_3 r_3$ $*$
aR R_4 r_4 $*$ aR R_5 r_5 $*$ aR R_6 r_6 $*$ aRR_7 r_7 $*$ aR R_8 r_8 $*$ aR R_9 r_9 $*$
aS1 psrA $ipsr.$A $*$ aS1 psrC $ipsr.$C $*$ aS1 psrI $ipsr.$I $*$
aS1 psrN $ipsr.$N $*$ aS1 psrQ $ipsr.$Q $*$ aS1 psrV $ipsr.$V $*$
$\neg$aU$_2$ $divisor\ 0\ cushion\ ie_{\mathrm{rls}}\ ie_{\mathrm{rda}}$ F $[]$ $\underline{tx\_running\_addr}\ df\ \underline{start_{\mathrm{tx}}}$
$\underline{size_{\mathrm{tx}}}\ \underline{front\_addr_{\mathrm{tx}}}\ \underline{back\_addr_{\mathrm{tx}}}\ df_{\mathrm{tx}}\ str_{\mathrm{tx}}\ spaceoe\ front_{\mathrm{tx}}$
$back_{\mathrm{tx}}\ str_{\mathrm{txout}}\ str_{\mathrm{txbuf}}\ f_{\mathrm{tx}}\ tx\_running\ \underline{start_{\mathrm{rx}}}\ \underline{size_{\mathrm{rx}}}\ \underline{front\_addr_{\mathrm{rx}}}\ \underline{back\_addr_{\mathrm{rx}}}$
$df_{\mathrm{rx}}\ strm_{\mathrm{rx0}}\ str_{\mathrm{rxbuf0}}\ front_{\mathrm{rx}}$
$*$ aPPSTACK irq $spirq$ $[]$ 15)

Figure 6.10: Hoare triple for the ISR in THRE-atomic mode

$$\vdash\ \mathsf{SPEC\ LPC\_IRQ\_MODEL} \qquad \text{(i)}$$

$$(\mathsf{aPC}\ 3200 * \mathsf{aR\ LRsvc}\ \mathit{lrsvc} * \qquad \text{(ii)}$$

$$\neg\mathsf{aPSR}\ \mathit{ips} * \neg\mathsf{aR\ LRirq}\ * \mathsf{aPPSTACK\ irq}\ \mathit{spirq}\ []\ 15 * \qquad \text{(iii)}$$

$$\left.\begin{array}{l}\mathsf{aMD}\ 19\ * \mathsf{aCPSR}\ \mathit{cpsr}\ * \mathsf{aS1\ psrA}\ \mathit{psra} * \mathsf{aS1\ psrC}\ \mathit{psrc} * \mathsf{aS1\ psrI\ F} * \\ \mathsf{aS1\ psrN}\ \mathit{psrn} * \mathsf{aS1\ psrQ}\ \mathit{psrq} * \mathsf{aS1\ psrV}\ \mathit{psrv}\ * \mathsf{aS1\ psrZ}\ \mathit{psrz} *\end{array}\right\} \qquad \text{(iv)}$$

$$\mathsf{aPSTACK\ svc}\ \mathit{spsvc}[][w_1; w_3; w_5; w_7; w_9] * \qquad \text{(v)}$$

$$\left.\begin{array}{l}\mathsf{aR}\ \mathsf{R}_0\ r_0 * \mathsf{aR}\ \mathsf{R}_1\ r_1 * \mathsf{aR}\ \mathsf{R}_2\ r_2 * \mathsf{aR}\ \mathsf{R}_3\ r_3 * \mathsf{aR}\ \mathsf{R}_4\ r_4 * \\ \mathsf{aR}\ \mathsf{R}_5\ r_5 * \mathsf{aR}\ \mathsf{R}_6\ r_6 * \mathsf{aR}\ \mathsf{R}_7\ r_7 * \mathsf{aR}\ \mathsf{R}_8\ r_8 * \\ \mathsf{aR}\ \mathsf{R}_9\ r_9 * \mathsf{aR}\ \mathsf{R}_{10}\ r_{10} * \mathsf{aR}\ \mathsf{R}_{11}\ r_{11} * \mathsf{aR}\ \mathsf{R}_{12}\ r_{12} *\end{array}\right\} \qquad \text{(vi)}$$

$$\left.\begin{array}{l}\neg\mathsf{aU}_8\ \mathit{divisor}\ 97\ 97\ \mathit{cushion}\ \mathit{ie}_{\mathrm{rls}}\ \mathit{ie}_{\mathrm{rda}}\ \mathit{ie}_{\mathrm{thre}}\ \mathit{df}\ \mathit{df}_{\mathrm{rx}}\ \mathit{df}_{\mathrm{tx}}\ \underline{\mathit{running\_addr}_{\mathrm{tx}}} \\ \underline{\mathit{front\_addr}_{\mathrm{tx}}}\ \underline{\mathit{back\_addr}_{\mathrm{tx}}}\ \underline{\mathit{front\_addr}_{\mathrm{rx}}}\ \underline{\mathit{back\_addr}_{\mathrm{rx}}}\ \underline{\mathit{start}_{\mathrm{tx}}} \\ \underline{\mathit{size}_{\mathrm{tx}}}\ \underline{\mathit{start}_{\mathrm{rx}}}\ \underline{\mathit{size}_{\mathrm{rx}}}\ \mathit{str}_{\mathrm{tx}}\ 1\ \mathit{strm}_{\mathrm{rx0}}\ \mathit{str}_{\mathrm{rxbuf0}}\ \mathit{front}_{\mathrm{rx}} *\end{array}\right\} \qquad \text{(vii)}$$

$$\left.\begin{array}{l}\mathsf{cond}\ ((3\ \&\&\ \mathit{lrsvc} = 0) \wedge (3\ \&\&\ \mathit{spirq} - 56 = 0) \wedge \\ (3\ \&\&\ \mathit{spirq} - 60 = 0) \wedge (3\ \&\&\ \mathit{spsvc} - 20 = 0)))\end{array}\right\} \qquad \text{(viii)}$$

$$\mathsf{isr\_code} \cup \mathsf{driver\_code} \qquad \text{(ix)}$$

$$(\mathsf{aPC}\ \mathit{lrsvc} * \mathsf{aR\ LRsvc}\ 3336 * \qquad \text{(x)}$$

$$\neg\mathsf{aPSR}\ \mathit{ips} * \neg\mathsf{aR\ LRirq} * \mathsf{aPPSTACK\ irq}\ \mathit{spirq}\ []\ 15 * \qquad \text{(xi)}$$

$$\left.\begin{array}{l}\mathsf{aMD}\ 19 * \mathsf{aCPSR}\ \mathit{cpsr} * \mathsf{aS1\ psrA}\ \mathit{psra} * \neg\mathsf{aS1\ psrC} * \mathsf{aS1\ psrI\ F} * \\ \neg\mathsf{aS1\ psrN} * \mathsf{aS1\ psrQ}\ \mathit{psrq} * \mathsf{aS1\ psrV\ F} * \neg\mathsf{aS1\ psrZ} *\end{array}\right\} \qquad \text{(xii)}$$

$$\mathsf{aPSTACK\ svc}\ \mathit{spsvc}\ []\ [\mathit{lrsvc}; r_7; r_6; r_5; r_4]) \qquad \text{(xiii)}$$

$$\left.\begin{array}{l}\mathsf{aR}\ \mathsf{R}_0\ (255\ \ \&\&\ \ r_0) * \neg\mathsf{aR}\ \mathsf{R}_1 * \neg\mathsf{aR}\ \mathsf{R}_2 * \neg\mathsf{aR}\ \mathsf{R}_3 * \mathsf{aR}\ \mathsf{R}_4\ r_4 * \\ \mathsf{aR}\ \mathsf{R}_5\ r_5 * \mathsf{aR}\ \mathsf{R}_6\ r_6 * \mathsf{aR}\ \mathsf{R}_7\ r_7 * \mathsf{aR}\ \mathsf{R}_8\ r_8 * \\ \mathsf{aR}\ \mathsf{R}_9\ r_9 * \mathsf{aR}\ \mathsf{R}_{10}\ r_{10} * \mathsf{aR}\ \mathsf{R}_{11}\ r_{11} * \mathsf{aR}\ \mathsf{R}_{12}\ r_{12} *\end{array}\right\} \qquad \text{(xiv)}$$

$$\left.\begin{array}{l}\neg\mathsf{aU}_8\ \mathit{divisor}\ 97\ 97\ \mathit{cushion}\ \mathit{ie}_{\mathrm{rls}}\ \mathit{ie}_{\mathrm{rda}}\ \mathsf{T}\ \mathit{df}\ \mathit{df}_{\mathrm{rx}}\ \mathit{df}_{\mathrm{tx}}\ \underline{\mathit{running\_addr}_{\mathrm{tx}}} \\ \underline{\mathit{front\_addr}_{\mathrm{tx}}}\ \underline{\mathit{back\_addr}_{\mathrm{tx}}}\ \underline{\mathit{front\_addr}_{\mathrm{rx}}}\ \underline{\mathit{back\_addr}_{\mathrm{rx}}}\ \underline{\mathit{start}_{\mathrm{tx}}} \\ \underline{\mathit{size}_{\mathrm{tx}}}\ \underline{\mathit{start}_{\mathrm{rx}}}\ \underline{\mathit{size}_{\mathrm{rx}}}\ (r_0 \uparrow^{8}_{32} :: \mathit{str}_{\mathrm{tx}})\ 0\ \mathit{strm}_{\mathrm{rx0}}\ \mathit{str}_{\mathrm{rxbuf0}}\ \mathit{front}_{\mathrm{rx}}\end{array}\right\} \qquad \text{(xv)}$$

Figure 6.11: Hoare triple for the working branch of uart0Putch

Table 6.3: Placeholders used in the Hoare triple for `uart0Putch`

constant	description
$start_{\mathrm{rx}}$	0x400001A8
$size_{\mathrm{rx}}$	64
$start_{\mathrm{tx}}$	0x40000010
$size_{\mathrm{tx}}$	128
$front\_addr_{\mathrm{rx}}$	0x400001EA
$back\_addr_{\mathrm{rx}}$	0x40000194
$front\_addr_{\mathrm{tx}}$	0x4000019C
$back\_addr_{\mathrm{tx}}$	0x40000092
$front_{\mathrm{rx}}$	$f \nearrow^{16}_{2}$ 0x400001EA
$back_{\mathrm{rx}}$	$f \nearrow^{16}_{2}$ 0x40000194
$front_{\mathrm{tx}}$	$f \nearrow^{16}_{2}$ 0x4000019C
$back_{\mathrm{tx}}$	$f \nearrow^{16}_{2}$ 0x40000092
$running\_addr_{\mathrm{tx}}$	$f \nearrow^{16}_{4}$ 0x400001A4

CHAPTER 7

CONCLUSION

In this research I have shown that full correctness of realistic interrupt-driven drivers for realistic devices can be proven. I introduced my approach and demonstrated it by proving the full correctness of a serial port driver. Below is the summary of the approach.

Model the device by describing its own transition and the effects of memory-mapped access by the processor core at each clock cycle. The clock cycle is internal to the device and implements a clock divider. The device runs in lock-step with the processor. Its speed is parameterized relative to the processor core. This allows timing constraints to be expressed.

The asymmetry between the ISR and the main program is leveraged. The role of the ISR is viewed as maintaining some invariants for the resources shared between the main program and itself. Changes to these invariants only occur when related interrupts are disabled. ISR is verified first, then its effect can be integrated on the semantics of the instructions of the main program. The main program can be verified using the approach for sequential programs.

Automation can be achieved in the work flow when high-level assertions are used to describe the semantics of the device and instructions. Models and theorems about common data structures should be treated as an infrastructure, just like the ARM model. I spent significant time to formalize the circular buffer and prove lemmas about it. If this work had been done, I could have saved quite some time.

CHAPTER 8

FUTURE WORK

I have demonstrated that full correctness of realistic drivers for a realistic device can be proven. In this chapter I explain some possible ways to further this research.

8.1 Multiple Devices

In the system I verified, there is only one device. In real embedded systems, more than one device is expected. It would be interesting to extend the approach to drivers of multiple devices. Multiple devices are already supported to some extent in the abstract device model. The model supports combination of multiple devices. However, this is limited to polling-based drivers.

If all the device drivers are polling-based, the verification is a trivial repetition of the approach in this dissertation on each device and its drivers. However, it is far more likely that device drivers in a system with multiple devices are interrupt-driven. Such drivers utilize the processor and devices more efficiently and responsively.

To continue this work on ARM, the first step is to formalize a vector interrupt controller (VIC). A VIC itself is a device. One has to be careful when combining a VIC model with other device models, as the VIC needs to be configured to support multiple interrupts from these devices. One way is to revise the device combination design to accommodate the special VIC device. The better way is to design a wellformness constraint to assert that the VIC is properly configured for other devices.

In the verification of the serial port driver, the interrupt handler is verified before the main program. Since there is only one device and there is no VIC, the handler is equivalent to the ISR. What is really verified is the interrupt handling process starting from when an interrupt is taken until the interrupted main program is resumed. With multiple devices, interrupt handling is more likely not done by one single ISR. These ISRs are organized by an interrupt handling scheme.

There are more than one choice for interrupt handling schemes. The most basic one is the nonnested scheme. In this scheme interrupts are handled in first come, first serve fashion. All the ISRs are not interrupted and they can be verified independently. The approach used in this dissertation needs to be repeated to every interrupt. This scheme is easy to verify. But it is possible that lower-priority interrupt can prevent higher-priority interrupts from being served.

The next one up is the nested scheme. When an interrupt is taken and has been processed for a while, interrupt handling is enabled. Generally, lower latency is achieved. But interrupts with higher priority may not have lower latency compared to interrupts with lower priority. This scheme involves sophisticated context switching. If interrupts come at too fast rate, stack overflow may occur. To prove termination and stack safety in interrupt handling in this scheme, timing constraint is necessary.

Further up priority filtering can be added to the nested scheme. It yields lower latency for higher-priority interrupts, but also adds more challenges for verification. One can start from the ISR with the highest priority and work down the priority hierarchy to prove the Hoare triples for the interrupt handling process.

If all devices are working together in an application, then fairness is a property needing to be verified. It is fair when lower-priority interrupts are served promptly. Again, timing constraints will be useful in verifying properties like this.

Theoretically, timing analysis for complicated interrupt handling schemes is essentially applying queuing and scheduling on the interrupts. It is hard, especially if it is done using a theorem prover. Conclusions from independent timing analysis may have to be introduced into the theorem prover as oracles. This may compromise the soundness, but makes the verification scale.

Regehr [77] surveyed the problems that developers face when creating correct interrupt-driven embedded software, and proposed some guidelines. Some of these guidelines are related to the timing constraints. For example, the maximum arrival rate, the deadline, and WCET for each interrupt should be determined. Following these guidelines make the verification easier, especially the timing analysis.

A system with a serial port, an analog-to-digital converter (ADC) and a timer is a good example. It can be used to collect and transmit data. This system has inherent timing constraints, yet is simple enough for verification purpose. The ADC and timer need to be formalized. Similar to the serial port model, these models should have internal clocks to define their relative speeds to the processor.

8.2 Initialization Code

The work on serial port drivers shows that my approach can be used to verify programs which allow disabling and enabling hardware interrupts. Following the verification of drivers of multiple devices, a good choice for the next step in future work is verifying the initialization code of a simple embedded system.

The research presented in this dissertation assumes the processor and devices are in wellformed states. These wellformness are expressed in terms of global invariants. The initialization process will establish invariants. When an embedded system is powered on, it first reads and executes the code from a known address in the read-only memory (ROM). This code then jumps to the code in program memory, which then initializes processor registers, stacks, memory and cache. It then disables interrupts.

Since RAM is faster than program memory, the code will copy the data sections from program memory to the RAM. After this, devices are initialized.

ROM, program memory and RAM can all be modeled as devices, so this bootstrapping process can be verified. Cache interferes with timing analysis. Independent results about it can be used.

The serial port model presented in this dissertation has a rather strong wellness constraint, which describes the working state of the device. To verify the device initialization process is to prove that such wellformness constraints can be established by the initialization code. This will test some parts of the device models which are usually not touched in verifications.

CHAPTER 9

RELATED WORK

In this chapter I discuss some of the recent related work, ranging from formal semantics for ISA, program logic for machine languages, some examples of verification on machine language level, handling interrupts-caused concurrency, as well as verification of device drivers.

9.1 Formal Semantics of ISAs

The device model presented in this work is intended to be integrated with an instruction set architecture (ISA) model. There has been some work on formal models of an industrial ISA. Boyer and Yu [12] formalized a substantial subset of Motorola MC68020, and verified compiler generated machine code programs using it. In the process they proved a large number of lemmas to achieve automation. The work is done in Nqthm, also known as Boyer-Moore theorem prover [11].

For ARM architecture, Fox from the University of Cambridge has formalized a series of ARM ISAs. ARM6 [32, 33] was first formalized and verified using an algebraic approach [34], where the state function was modeled as an iterated map. My work on the correctness of polling-based serial port driver in Chapter 3 uses this ARM6 model. Later, Fox and Myreen formalized ARMv7 [37] using a monadic approach. This model covers all the currently supported ARM versions. An overview of the ARM specification and verification using HOL is done by Fox et al. [36].

For x86 architecture, Hunt formalized Centaur Technology x86 ISA [98] using E-language which is deeply embedded in ACL2 [46]. However, this work is not publicly available. Currently, there is ongoing effort [97] to formalize x86 in ACL2.

Sarkar et al. formalized the operational semantics of x86 multiprocessor machine code with so called causal consistency (CC) memory model [85] in HOL4 [90]. The CC model is unsound with regard to the hardware and too weak for programming.

Later, the total store ordering (TSO) model for x86 machine code semantics was developed [69] to fix the issues.

Degenbaev [23] also formalized the x86 ISA, including the memory model and the operational semantics. He extended the TSO memory model with caches, translation-lookaside buffers, memory fences, locks, and other features of the x86 processor. He also designed a new domain-specific monadic language to specify the instruction semantics. However, the semantics specification is not inside a theorem prover.

For IBM Power architecture, Alglave et al. formalized Power multiprocessor semantics model [1] in HOL4. However, some subtle issues of the memory behavior were not accounted for. This was later addressed in an operational memory model [84] and an axiomatic memory model [51]. For the latter, a SAT solver is used to evaluate the specification with great performance.

9.2 Program Logic for Machine Languages

Hoare logic [31, 42] pioneered the way to mathematically reason about the meaning of a program. There has been research on how to use it on assembly programs. One direction is to support the low-level control flow structure. Tan and Appel [93] designed a program logic that is able to verify programs with the control flow constructs typical to machine languages, such as the multientry and multiexit constructs. They used it to verify a type system with the SPARC machine language. This type system is used to prove memory safety of the machine code program. Using ACL2, Hardin et al. [41] used a cutpoint composition [52] to prove secure machine code application. Especially, cutpoint is used to prove partial correctness of loop. They formalized Rockwell Collins's AAMP7G architecture in the process. Saabas and Uustalu [83] designed a simple low-level language with jumps and derived a compositional semantics and Hoare logic for it.

The Flint group at Yale University has done a broad range of work along the certified assembly program (CAP) project to tackle issues in verifying assembly programs using Coq [43], such as dynamic storage allocation [102],concurrency with non-preemptive threads [28, 103], embedded code pointers [66], machine context management [67] stack-based controls [30],garbage collection [50, 53], self-modifying code [16], interrupts and preemptive threads [29]. Most of these works are in the context of proof-carrying code [65]. Assume-guarantee [55] or rely-guarantee [45] is used in concurrency and interrupt related work.

Out of these works, the one on interrupts and preemptive threads[29] is mostly related to my work. The authors developed a program logic for machine languages with support for interrupts and threads. It has the assume-guarantee style. The memory shared between a thread and the ISRs is reasoned about using the so-called ownership-transfer semantics, which is inspired by the concurrent separation logic [13]. Even though this work supports interrupts, it does not support devices.

Myreen and Gordon [61] developed a separation logic style Hoare logic for realistic machine programs in HOL4. Local reasoning was supported using the frame rule. It supports the low-level features such as the same memory space for data and the program, but peripheral devices and interrupts are not supported. Later this work was extended to work with multiple architectures in decompiling [62, 63]. With this foundation, Myreen and others were able to do some significant verification, such as proof-producing compilation [64], just-in-time compiler for x86 [58], a verified runtime for a verified theorem prover [59], verified LISP implementations on ARM, x86 and PowerPC [89], translation validation of the seL4 from C to ARM level [89]. My work is based on this work without using the decompilation.

Chlipala [17] implemented a computational separation logic to achieve much automation in verifying low-level programs. It is called computational because the function specifications are mostly written in terms of reference implementations in the pure functional language inside Coq.

When dealing with concurrency related to threads or interrupts, researchers often borrow ideas from assume-guarantee [55], or rely-guarantee [45] and concurrent separation logic [80]. This is evident in the work from the Flint Group at Yale University, mentioned above. They actually investigated the relationship between concurrent separation logic and assume-guarantee method based on assembly language [27]. They proposed a assume-guarantee based logic for an assembly language with built-in lock primitives, and showed that concurrent separation logic is a specialization of this logic. Recently, Ridge [81] used a rely-guarantee style approach to prove x86 assembly program running under the TSO memory model [69]. It is formalized in

HOL4. Fu et al. [38] adapted concurrent separation logic to verify concurrent assembly programs. They extended the lock in concurrent separation logic so that it can track the reentrancy level. It is formalized in Coq.

My treatment of the concurrency caused by interrupt also borrowed ideas from assume-guarantee and concurrent separation logic. On one hand, since the Hoare logic I used provides complete correctness, the interrupt handling has clear entry and exit interface with the main program. It is similar to the lock in concurrent separation logic. On the other hand, the assertion formula for the main program and the ISR are constructed in the spirit of assume-guarantee method.

9.3 Taming Concurrency in Interrupt-Driven Systems

Parallelism introduced by interrupt handling to interrupt-driven embedded systems presents a big challenge for verifying them. Timing-related assumptions or design choice often are made to make verification feasible. Kotker et al. [48] did timing analysis of interrupt-driven embedded systems using context bounds technique [75, 76]. The idea is to bound the number of context switches, and use a sequential program to simulate all interleaving of threads inside the bound. Brylow et al. [14] assumed worst-case execution time for program fragments when statically checking the deadline properties of interrupt-driven embedded systems. The assumptions serve as oracles for static analysis and need testing to verifying them. In the verification of seL4 kernel [47], interrupt handling is restricted to certain convenient windows in the design of the kernel.

Model checking needs to address similar issues. Schlich et al [88] used an abstraction technique based on partial order reduction to reduce the state space explosion when using model checking to verify interrupt-driven embedded systems. The insight is that if the ISR does not have an effect on the behavior of the main program, then its execution can be blocked. This can be statically done.

In contrast, I use timing constraints to help prove the termination of the ISR and to achieve more precision of the specification of the ISR. Interleaving between the main program and the ISR is removed at the instruction level when the effect of the ISR is integrated. My approach allows the techniques for verifying sequential program to be used.

9.4 Formal Methods on Interrupt-driven Embedded Systems

Formal methods have been applied on interrupts-driven systems. Most of the time the approach is only intended to tackle a small subset of problems. Regehr and Cooperider [78] discussed the approach of verifying interrupts through verifying threads that have the same semantics as the interrupt handlers. This approach can take advantage of the existing techniques of threads verification. However, it is difficult to introduce notion of time with this approach.

Palsberg et al. [70] proposed a typed interrupt calculus for programming interrupt-driven systems. The success of type check would guarantee stack boundness. Regehr et al. [79] uses context-sensitive dataflow analysis of object code to statically guarantee stack safety in interrupt-driven embedded software. In my work, stack safety is assumed in the precondition. The size of the stack is automatically calculated. Suenaga and Kobayashi [91] developed a concurrent calculus with primitives for threads and interrupts. It uses type system to guarantee deadlock freedom.

Zhao et al. [104] constructed a formal probabilistic language to describe the operational semantics for an interrupt-driven program. The delay on the main program due to interrupts handling can be specified. The timing constraints in my work are more concerned with the delay on the ISR from the main program.

9.5 Verification of Device Drivers

Device drivers are typically written in unsafe programming languages and live in the kernel's address space. Driver bugs can corrupt or drop data, cause peripherals to malfunction or become wedged, and crash the operating system [18].

9.5.1 Correctness of Device Drivers

Monniaux [56] verified a driver together with a model for a USB (universal serial bus) OHCI (open host controller interface) controller. This work noticed that a driver should be verified together with the model for the device. The controller was modeled as a C program, which runs in parallel with the driver. A static analyzer was used to do the verification. The parallelism between the driver and the device model is reduced using partial order reduction. The execution of the device model is only considered at the point where the driver accesses the shared memory. This is possible because timing constraints are not considered in the device model.

Alkassar and his colleagues have investigated the correctness of a device driver using Isabelle/HOL [68] in the context of the Verisoft project [94]. They constructed a formal model for a serial port device and verified a serial port driver [2]. The driver is written in DLX instructions of the VAMP (verified architecture microprocessor processor) [9]. The models of the processor and the device were combined in a lockstep fashion. The serial port model is similar to mine, but without the notion of timing. As a result, the problem of overrunning the receiver queue cannot be expressed. Later, they verified a hard disk driver [3]. The interleaving between the device model and the driver is reduced by reordering. This idea is also used by Monniaux [56]. They reported a formal pervasive verification of an operating system kernel including inline assembly, C program and concurrent programs [4]. Again, the interleaving steps between the processor and devices are reordered to have a sequential semantics. In contrast, we modeled the relative speed of the serial port device so that some timing properties can be reasoned about; the accuracy depends on the details of the instruction set architecture model that is used.

9.5.2 Property Checking for Device Drivers

For desktop operating systems such as Windows and Linux, full correctness proof of device drivers are not realistic. Instead, research has focused on property checking and bug finding for device drivers for these operating systems. The main goal of most of these works are to keep drivers from crashing or hanging the operating system.

There has been great success at using CEGAR (counterexample guided abstraction refinement) [20] based model checking to verify device drivers for commodity operating systems [5, 6, 7, 74, 100]. Most of this work is at the source code level and focuses on the interface between the drivers and the kernel, i.e., API rule, as opposed to focusing on correct interaction with the device.

Recent tools can check memory safety. SLAYer [8] automatically tries to prove memory safety in industrial system code. It is based on separation logic and Z3. The main limitation is that it uses an ideal memory model in the sense that memory is a collection of disjointed objects, so it is not byte-level accurate. Penninckx et al. reported a verification of a realistic Linux USB keyboard driver using the VeriFast tool [71]. VeriFast [44] can check memory safety and API rules violations. It works on annotated C programs. The annotation is based on separation logic involving ghost variables and ghost lemmas.

Source level model checking can be difficult to use in the context of embedded systems [87]. Model checking of embedded C code [26] and assembly code [25, 86] has been done. In these works, specific hardware details are considered. However, the works are largely limited to bug hunting instead of providing correctness guarantees.

9.5.3 Device Driver Synthesis

One approach to improve the reliability of device drivers is to mechanically generate "correct by construction" drivers from a high level formal specification of a device and its environment [21, 54, 82, 92, 96]. By avoiding languages like C and by checking some properties, bugs can be avoided. This approach has advantages, such as making it easier to generate drivers for multiple platforms. However, the resulting driver is not verified (the code generator and compiler are trusted) and synthesis of high-performance drivers remains challenging.

REFERENCES

[1] Alglave, J., Fox, A. C. J., Ishtiaq, S., Myreen, M. O., Sarkar, S., Sewell, P., and Nardelli, F. Z. 2009. The semantics of Power and ARM multiprocessor machine code. In *Proceedings of the POPL 2009 Workshop on Declarative Aspects of Multicore Programming*, Savannah, GA, Jan. 2009, L. Petersen and M. M. T. Chakravarty, Eds. ACM, New York, NY, 13–24.

[2] Alkassar, E., Hillebrand, M., Knapp, S., Rusev, R., and Tverdyshev, S. 2007. Formal device and programming model for a serial interface. In *Proceedings of the 4th International Verification Workshop (VERIFY)*, Bremen, Germany, July 2007, B. Beckert, Ed. CEUR-WS.org, 4–20.

[3] Alkassar, E. and Hillebrand, M. A. 2008. Formal functional verification of device drivers. In *Proceedings of the 2nd International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, Toronto, Canada, Oct. 2008, N. Shankar and J. Woodcock, Eds. Springer, Berlin, Germany, 225–239.

[4] Alkassar, E., Paul, W. J., Starostin, A., and Tsyban, A. 2010. Pervasive verification of an OS microkernel - Inline assembly, memory consumption, concurrent devices. In *Proceedings of the 3rd International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, Edinburgh, UK, Aug. 2010, G. T. Leavens, P. W. O'Hearn, and S. K. Rajamani, Eds. Springer, Berlin, Germany, 71–85.

[5] Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S. K., and Ustuner, A. 2006. Thorough static analysis of device drivers. In *Proceedings of the 2006 EuroSys Conference*, Leuven, Belgium, Apr. 2006, Y. Berbers and W. Zwaenepoel, Eds. ACM, New York, NY, 73–85.

[6] Ball, T., Bounimova, E., Kumar, R., and Levin, V. 2010. SLAM2: Static driver verification with under 4% false alarms. In *Proceedings of the 2010 Formal Methods in Computer-Aided Design (FMCAD)*, Lugano, Switzerland, Oct. 2010, R. Bloem and N. Sharygina, Eds. FMCAD Inc, Austin, TX, 35–42.

[7] Ball, T. and Rajamani, S. K. 2001. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th International SPIN Workshop on Model Checking Software (SPIN)*, Toronto, Canada, May 2001, M. B. Dwyer, Ed. Springer, Berlin, Germany, 103–122.

[8] Berdine, J., Cook, B., and Ishtiaq, S. 2011. Slayer: Memory safety for systems-level code. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, Snowbird, UT, July 2011, G. Gopalakrishnan and S. Qadeer, Eds. Springer, Berlin, Germany, 178–183.

[9] Beyer, S., Jacobi, C., Kröning, D., Leinenbach, D., and Paul, W. J. 2006. Putting it all together - Formal verification of the VAMP. *International Journal on Software Tools for Technology Transfer 8*, 411–430.

[10] Böhme, S., Fox, A. C. J., Sewell, T., and Weber, T. 2011. Reconstruction of Z3's bit-vector proofs in HOL4 and Isabelle/HOL. In *Proceedings of the 1st Certified Programs and Proofs (CPP)*, Kenting, Taiwan, Dec. 2011, J.-P. Jouannaud and Z. Shao, Eds. Springer, Berlin, Germany, 183–198.

[11] Boyer, R. S., Kaufmann, M., and Moore, J. S. 1995. The Boyer-Moore theorem prover and its interactive enhancement. *Computers & Mathematics with Applications 29*, 27–62.

[12] Boyer, R. S. and Yu, Y. 1992. Automated correctness proofs of machine code programs for a commercial microprocessor. In *Proceedings of the 11th International Conference on Automated Deduction*, Saratoga Springs, NY, June 1992, D. Kapur, Ed. Springer, Berlin, Germany, 416–430.

[13] Brookes, S. 2007. A semantics for concurrent separation logic. *Theoretical Computer Science 375*, 227–270.

[14] Brylow, D. and Palsberg, J. 2003. Deadline analysis of interrupt driven software. In *Proceedings of the 11th International Symposium on the Foundations of Software Engineering (FSE)*, Helsinki, Finland, Sept. 2003, J. Paakki and P. Inverardi, Eds. ACM, New York, NY, 198–207.

[15] Burstall, R. M. 1972. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence 7*, 23–50.

[16] Cai, H., Shao, Z., and Vaynberg, A. 2007. Certified self-modifying code. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2007, J. Ferrante and K. S. McKinley, Eds. ACM, New York, NY, 66–77.

[17] Chlipala, A. 2011. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the ACM SIGPLAN 2011 Conference on Programming Language Design and Implementation (PLDI)*, San Jose, CA, June 2011, M. W. Hall and D. A. Padua, Eds. ACM, New York, NY, 234–245.

[18] Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. R. 2001. An empirical study of operating system errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, Oct. 2001, K. Marzullo and M. Satyanarayanan, Eds. ACM, New York, NY, 73–88.

[19] Clarke, E. M., Emerson, E. A., and Sistla, A. P. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems 8*, 244–263.

[20] Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., and Veith, H. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM 50*, 752–794.

[21] Conway, C. L. and Edwards, S. A. 2004. NDL: A domain-specific language for device drivers. In *Proceedings of the 2004 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Washington, DC, June 2004, D. B. Whalley and R. Cytron, Eds. ACM, New York, NY, 30–36.

[22] De Sutter, B., Van Put, L., Chanet, D., De Bus, B., and De Bosschere, K. 2007. Link-time compaction and optimization of ARM executables. *ACM Transactions on Embedded Computing Systems (TECS) 6*, 5.

[23] Degenbaev, U. 2011. *Formal Specification of the x86 Instruction Set Architecture.* Ph.D. Dissertation, Saarland University, Saarbrcken.

[24] Emerson, E. A. and Halpern, J. Y. 1985. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences 30*, 1–24.

[25] Fehnker, A., Huuck, R., Rauch, F., and Seefried, S. 2008. Some assembly required - Program analysis of embedded system code. In *Proceedings of the 8th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Beijing, China, Sept. 2008, J. R. Cordy and L. Zhang, Eds. IEEE Computer Society, Washington, DC, 15–24.

[26] Fehnker, A., Huuck, R., Schlich, B., and Tapp, M. 2009. Automatic bug detection in microcontroller software by static program analysis. In *Proceedings of the 35th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, Spindleruv Mlýn, Czech Republic, Jan. 2009, M. Nielsen, A. Kucera, P. B. Miltersen, C. Palamidessi, P. Tuma, and F. D. Valencia, Eds. Springer, Berlin, Germany, 267–278.

[27] Feng, X., Ferreira, R., and Shao, Z. 2007. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *Proceedings of the 16th European Symposium on Programming (ESOP)*, Braga, Portugal, Mar.–Apr. 2007, R. De Nicola, Ed. Springer, Berlin, Germany, 173–188.

[28] Feng, X. and Shao, Z. 2005. Modular verification of concurrent assembly code with dynamic thread creation and termination. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Tallinn, Estonia, Sept. 2005, O. Danvy and B. C. Pierce, Eds. ACM, New York, NY, 254–267.

[29] Feng, X., Shao, Z., Guo, Y., and Dong, Y. 2009. Certifying low-level programs with hardware interrupts and preemptive threads. *Journal of Automatic Reasoning 42*, 301–347.

[30] Feng, X., Shao, Z., Vaynberg, A., Xiang, S., and Ni, Z. 2006. Modular verification of assembly code with stack-based control abstractions. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI)*, Ottawa, Canada, June 2006, M. I. Schwartzbach and T. Ball, Eds. ACM, New York, NY, 401–414.

[31] Floyd, R. W. 1966. Assigning meanings to programs. In *Proceedings of Symposium in Applied Mathematics*, New York, NY, Apr. 1966. Vol. 19. American Mathematical Society, Providence, RI, 19–32.

[32] Fox, A. C. J. 2002. Formal verification of the ARM6 micro-architecture. Tech. Rep. 548, University of Cambridge Computer Laboratory. Nov.

[33] Fox, A. C. J. 2003. Formal specification and verification of ARM6. In *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, Rome, Italy, Sept. 2003, D. A. Basin and B. Wolff, Eds. Springer, Berlin, Germany, 25–40.

[34] Fox, A. C. J. 2005. An algebraic framework for verifying the correctness of hardware with input and output: A formalization in HOL. In *Proceedings of the 1st International Conference on Algebra and Coalgebra in Computer Science (CALCO)*, Swansea, UK, Sept. 2005, J. L. Fiadeiro, N. Harman, M. Roggenbach, and J. J. M. M. Rutten, Eds. Springer, Berlin, Germany, 157–174.

[35] Fox, A. C. J. 2011. LCF-style bit-blasting in HOL4. In *Proceedings of the 2nd International Conference on Interactive Theorem Proving (ITP)*, Berg en Dal, The Netherlands, Aug. 2011, M. C. J. D. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, Eds. Springer, Berlin, Germany, 357–362.

[36] Fox, A. C. J., Gordon, M. J. C., and Myreen, M. O. 2010. Specification and verification of ARM hardware and software. In *Design and Verification of Microprocessor Systems for High-Assurance Applications*, D. S. Hardin, Ed. Springer, Berlin, Germany, 221–248.

[37] Fox, A. C. J. and Myreen, M. O. 2010. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Proceedings of the 2st International Conference on Interactive Theorem Proving (ITP)*, Edinburgh, UK, July 2010, M. Kaufmann and L. C. Paulson, Eds. Springer, Berlin, Germany, 243–258.

[38] Fu, M., Zhang, Y., and Li, Y. 2009. Formal reasoning about concurrent assembly code with reentrant locks. In *Proceedings of the 3rd IEEE Symposium on Theoretical Aspects of Software Engineering (TASE)*, Tianjin, China, July 2009, W.-N. Chin and S. Qin, Eds. IEEE Computer Society, Washington, DC, 233–240.

[39] Gordon, M. J. C. and Melham, T. F., Eds. 1993. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, Cambridge, UK.

[40] Gordon, M. J. C., Milner, R., and Wadsworth, C. P. 1979. *Edinburgh LCF: A Mechanized Logic of Computation.* Lecture Notes in Computer Science, vol. 78. Springer, Berlin, Germany.

[41] Hardin, D. S., Smith, E. W., and Young, W. D. 2006. A robust machine code proof framework for highly secure applications. In *Proceedings of the 6th International Workshop on the ACL2*, Seattle, WA, Aug. 2006, P. Manolios and M. Wilding, Eds. ACM, New York, NY, 11–20.

[42] Hoare, C. A. R. 1969. An axiomatic basis for computer programming. *Communications of the ACM 12*, 576–583.

[43] INRIA. A short introduction to Coq. Retrieved May 2013 from http://coq.inria.fr/a-short-introduction-to-coq/.

[44] Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., and Piessens, F. 2011. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *Proceedings of the 3rd NASA Formal Methods International Symposium (NFM)*, Pasadena, CA, Apr. 2011, M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds. Springer, Berlin, Germany, 41–55.

[45] Jones, C. B. 1983. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems 5*, 596–619.

[46] Kaufmann, M. and Moore, J. S. 1997. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering 23*, 203–213.

[47] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct. 2009, J. N. Matthews and T. E. Anderson, Eds. ACM, New York, NY, 207–220.

[48] Kotker, J., Sadigh, D., and Seshia, S. A. 2011. Timing analysis of interrupt-driven programs under context bounds. In *Proceedings of the 2011 Formal Methods in Computer-Aided Design (FMCAD)*, Austin, TX, Oct. 2011, P. Bjesse and A. Slobodová, Eds. FMCAD Inc, Austin, TX, 81–90.

[49] Lamport, L. 1994. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems 16*, 872–923.

[50] Lin, C., McCreight, A., Shao, Z., Chen, Y., and Guo, Y. 2007. Foundational typed assembly language with certified garbage collection. In *Proceedings of the 1st IEEE Symposium on Theoretical Aspects of Software Engineering (TASE)*, Shanghai, China, June 2007, J. He and J. Sanders, Eds. IEEE Computer Society, Washington, DC, 326–338.

[51] Mador-Haim, S., Maranget, L., Sarkar, S., Memarian, K., Alglave, J., Owens, S., Alur, R., Martin, M. M. K., Sewell, P., and Williams, D. 2012. An axiomatic memory model for POWER multiprocessors. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV)*, Berkeley, CA, July 2012, P. Madhusudan and S. A. Seshia, Eds. Springer, Berlin, Germany, 495–512.

[52] Matthews, J., Moore, J. S., Ray, S., and Vroon, D. 2006. Verification condition generation via theorem proving. In *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, Phnom Penh, Cambodia, Nov. 2006, M. Hermann and A. Voronkov, Eds. Springer, Berlin, Germany, 362–376.

[53] McCreight, A., Shao, Z., Lin, C., and Li, L. 2007. A general framework for certifying garbage collectors and their mutators. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2007, J. Ferrante and K. S. McKinley, Eds. ACM, New York, NY, 468–479.

[54] Mérillon, F. and Muller, G. 2001. Dealing with hardware in embedded software: A general framework based on the Devil language. In *Proceedings of the 2001 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES) / The Workshop on Optimization of Middleware and Distributed Systems (LCTES/OM)*, Snowbird, UT, June 2001, S. Hong and S. Pande, Eds. ACM, New York, NY, 121–127.

[55] Misra, J. and Chandy, K. M. 1981. Proofs of networks of processes. *IEEE Transactions on Software Engineering 7*, 417–426.

[56] Monniaux, D. 2007. Verification of device drivers and intelligent controllers: A case study. In *Proceedings of the 7th International Conference on Embedded Software (EMSOFT)*, Salzburg, Austria, Sept.–Oct. 2007, C. M. Kirsch and R. Wilhelm, Eds. ACM, New York, NY, 30–36.

[57] Moore, J. S. 2006. Inductive assertions and operational semantics. *International Journal on Software Tools for Technology Transfer 8*, 359–371.

[58] Myreen, M. O. 2010. Verified just-in-time compiler on x86. In *Proceedings of the 37th Symposium on Principles of Programming Languages (POPL)*, Madrid, Spain, Jan. 2010, M. V. Hermenegildo and J. Palsberg, Eds. ACM, New York, NY, 107–118.

[59] Myreen, M. O. and Davis, J. 2011. A verified runtime for a verified theorem prover. In *Proceedings of the 2nd International Conference on Interactive Theorem Proving (ITP)*, Berg en Dal, The Netherlands, Aug. 2011, M. C. J. D. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, Eds. Springer, Berlin, Germany, 265–280.

[60] Myreen, M. O., Fox, A. C. J., and Gordon, M. J. C. 2007. Hoare logic for ARM machine code. In *Proceedings of the 2007 Symposium on Fundamentals of Software Engineering (FSEN)*, Tehran, Iran, Apr. 2007, F. Arbab and M. Sirjani, Eds. Springer, Berlin, Germany, 272–286.

[61] Myreen, M. O. and Gordon, M. J. C. 2007. Hoare logic for realistically modelled machine code. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Braga, Portugal, Mar.–Apr. 2007, O. Grumberg and M. Huth, Eds. Springer, Berlin, Germany, 568–582.

[62] Myreen, M. O., Gordon, M. J. C., and Slind, K. 2008. Machine-code verification for multiple architectures - An application of decompilation into logic. In *Proceedings of the 2008 Formal Methods in Computer-Aided Design (FMCAD)*, Portland, Oregon, Nov. 2008, A. Cimatti and R. B. Jones, Eds. IEEE Computer Society, Washington, DC, 1–8.

[63] Myreen, M. O., Gordon, M. J. C., and Slind, K. 2012. Decompilation into logic - Improved. In *Proceedings of the 2012 Formal Methods in Computer-Aided Design (FMCAD)*, Cambridge, UK, Oct. 2012, G. Cabodi and S. Singh, Eds. FMCAD Inc, Austin, TX, 78–81.

[64] Myreen, M. O., Slind, K., and Gordon, M. J. C. 2009. Extensible proof-producing compilation. In *Proceedings of the 18th International Conference on Compiler Construction*, York, UK, Mar. 2009, O. de Moor and M. I. Schwartzbach, Eds. Springer, Berlin, Germany, 2–16.

[65] Necula, G. C. 1997. Proof-carrying code. In *Proceedings of the 24th Symposium on Principles of Programming Languages (POPL)*, Paris, France, Jan. 1997, N. D. Jones, Ed. ACM, New York, NY, 106–119.

[66] Ni, Z. and Shao, Z. 2006. Certified assembly programming with embedded code pointers. In *Proceedings of the 33rd Symposium on Principles of Programming Languages (POPL)*, Charleston, SC, Jan. 2006, J. G. Morrisett and S. L. P. Jones, Eds. ACM, New York, NY, 320–333.

[67] Ni, Z., Yu, D., and Shao, Z. 2007. Using XCAP to certify realistic systems code: Machine context management. In *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, Kaiserslautern, Germany, Aug. 2007, K. Schneider and J. Brandt, Eds. Springer, Berlin, Germany, 189–206.

[68] Nipkow, T., Paulson, L. C., and Wenzel, M. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic.* Lecture Notes in Computer Science, vol. 2283. Springer, Berlin, Germany.

[69] Owens, S., Sarkar, S., and Sewell, P. 2009. A better x86 memory model: x86-TSO. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, Munich, Germany, Aug. 2009, Z. Shao and B. C. Pierce, Eds. ACM, New York, NY, 391–407.

[70] Palsberg, J. and Ma, D. 2002. A typed interrupt calculus. In *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, Oldenburg, Germany, Sept. 2002, W. Damm and E.-R. Olderog, Eds. Springer, Berlin, Germany, 291–310.

[71] Penninckx, W., Mühlberg, J. T., Smans, J., Jacobs, B., and Piessens, F. 2012. Sound formal verification of Linux's USB BP keyboard driver. In *Proceedings of the 4th NASA Formal Methods International Symposium (NFM)*, Norfolk, VA, Apr. 2012, A. E. Goodloe and S. Person, Eds. Springer, Berlin, Germany, 210–215.

[72] Philips Semiconductors. 2004. LPC2119/2129/2194/2292/2294 user manual. Retrieved July 2009 from http://www.semiconductors.philips.com/acrobat/usermanuals/UM_LPC21XX_LPC22XX_2.pdf.

[73] Pnueli, A. 1977. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Scienc*, Providence, RI, Oct. 1977. IEEE Computer Society, Washington, DC, 46–57.

[74] Post, H. and Küchlin, W. 2007. Integrated static analysis for Linux device driver verification. In *Proceedings of the 6th International Conference on Integrated Formal Methods (IFM)*, Oxford, UK, July 2007, J. Davies and J. Gibbons, Eds. Springer, Berlin, Germany, 518–537.

[75] Qadeer, S. and Rehof, J. 2005. Context-bounded model checking of concurrent software. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Edinburgh, UK, Apr. 2005, N. Halbwachs and L. D. Zuck, Eds. Springer, Berlin, Germany, 93–107.

[76] Qadeer, S. and Wu, D. 2004. KISS: Keep it simple and sequential. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*, Washington, DC, June 2004, W. Pugh and C. Chambers, Eds. ACM, New York, NY, 14–24.

[77] Regehr, J. 2007. Safe and structured use of interrupts in real-time and embedded software. In *Handbook of Real-Time and Embedded Systems*, I. Lee, J. Y.-T. Leung, and S. Son, Eds. Chapman & Hall/CRC, New York, NY.

[78] Regehr, J. and Cooprider, N. 2007. Interrupt verification via thread verification. *Electronic Notes in Theoretical Computer Science 174*, 139–150.

[79] Regehr, J., Reid, A., and Webb, K. 2003. Eliminating stack overflow by abstract interpretation. In *Proceedings of the 3rd International Conference on Embedded Software (EMSOFT)*, Philadelphia, PA, Oct. 2003, R. Alur and I. Lee, Eds. Springer, Berlin, Germany, 306–322.

[80] Reynolds, J. C. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings of the17th IEEE Symposium on Logic in Computer Science (LICS)*, Copenhagen, Denmark, July 2002, G. Plotkin, Ed. IEEE Computer Society, Washington, DC, 55–74.

[81] Ridge, T. 2010. A rely-guarantee proof system for x86-TSO. In *Proceedings of the 3rd International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, Edinburgh, UK, Aug. 2010, G. T. Leavens, P. W. O'Hearn, and S. K. Rajamani, Eds. Springer, Berlin, Germany, 55–70.

[82] Ryzhyk, L., Chubb, P., Kuz, I., Sueur, E. L., and Heiser, G. 2009. Automatic device driver synthesis with Termite. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct. 2009, J. N. Matthews and T. E. Anderson, Eds. ACM, New York, NY, 73–86.

[83] Saabas, A. and Uustalu, T. 2006. A compositional natural semantics and Hoare logic for low-level languages. *Electronic Notes in Theoretic Computer Sciences 156*, 151–168.

[84] Sarkar, S., Sewell, P., Alglave, J., Maranget, L., and Williams, D. 2011. Understanding POWER multiprocessors. In *Proceedings of the ACM SIGPLAN 2011 Conference on Programming Language Design and Implementation (PLDI)*, San Jose, CA, June 2011, M. W. Hall and D. A. Padua, Eds. ACM, New York, NY, 175–186.

[85] Sarkar, S., Sewell, P., Nardelli, F. Z., Owens, S., Ridge, T., Braibant, T., Myreen, M. O., and Alglave, J. 2009. The semantics of x86-CC multiprocessor machine code. In *Proceedings of the 36th Symposium on Principles of Programming Languages (POPL)*, Savannah, GA, Jan. 2009, Z. Shao and B. C. Pierce, Eds. ACM, New York, NY, 379–391.

[86] Schlich, B. 2010. Model checking of software for microcontrollers. *ACM Transactions on Embedded Computing Systems (TECS) 9*, 36:1–36:27.

[87] Schlich, B. and Kowalewski, S. 2009. Model checking C source code for embedded systems. *International Journal on Software Tools for Technology Transfer 11*, 187–202.

[88] Schlich, B., Noll, T., Brauer, J., and Brutschy, L. 2009. Reduction of interrupt handler executions for model checking embedded software. In *Hardware and Software: Verification and Testing - 5th International Haifa Verification Conference (HVC)*, Haifa, Israel, Oct. 2009, K. S. Namjoshi, A. Zeller, and A. Ziv, Eds. Springer, Berlin, Germany, 5–20.

[89] Sewell, T. A. L., Myreen, M. O., and Klein, G. 2013. Translation validation for a verified OS kernel. In *Proceedings of the ACM SIGPLAN 2013 Conference on Programming Language Design and Implementation (PLDI)*, Seattle, WA, June 2013, H.-J. Boehm and C. Flanagan, Eds. ACM, New York, NY, 471–482.

[90] Slind, K. and Norrish, M. 2008. A brief overview of HOL4. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, Montreal, Canada, Aug. 2008, O. A. Mohamed, C. A. Muñoz, and S. Tahar, Eds. Springer, Berlin, Germany, 28–32.

[91] Suenaga, K. and Kobayashi, N. 2007. Type-based analysis of deadlock for a concurrent calculus with interrupts. In *Proceedings of the 16th European Symposium on Programming (ESOP)*, Braga, Portugal, Mar.–Apr. 2007, R. De Nicola, Ed. Springer, Berlin, Germany, 490–504.

[92] Sun, J., Yuan, W., Kallahalla, M., and Islam, N. 2005. HAIL: A language for easy and correct device access. In *Proceedings of the 2005 International Conference on Embedded Software (EMSOFT)*, Jersey City, NJ, Sept. 2005, W. Wolf, Ed. ACM, New York, NY, 1–9.

[93] Tan, G. and Appel, A. W. 2006. A compositional logic for control flow. In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Charleston, SC, Jan. 2006, E. A. Emerson and K. S. Namjoshi, Eds. Springer, Berlin, Germany, 80–94.

[94] The Verisoft Consortium. 2003. The Verisoft project. Retrieved May 2013 from http://www.verisoft.de/.

[95] Tuch, H., Klein, G., and Norrish, M. 2007. Types, bytes, and separation logic. In *Proceedings of the 34th Symposium on Principles of Programming Languages (POPL)*, Nice, France, Jan. 2007, M. Hofmann and M. Felleisen, Eds. ACM, New York, NY, 97–108.

[96] Wang, S. and Malik, S. 2003. Synthesizing operating system based device drivers in embedded systems. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Newport Beach, CA, Oct. 2003, R. Gupta, Y. Nakamura, A. Orailoglu, and P. H. Chou, Eds. ACM, New York, NY, 37–44.

[97] Warren A. Hunt, J. and Kaufmann, M. 2012. Towards a formal model of the x86 ISA. Tech. Rep. TR-12-07, University of Texas at Austin. Mar.

[98] Warren A. Hunt, J. and Swords, S. 2009. Centaur technology media unit verification. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV)*, Grenoble, France, July 2009, A. Bouajjani and O. Maler, Eds. Springer, Berlin, Germany, 353–367.

[99] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D. B., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P. P., Staschulat, J., and Stenström, P. 2008. The worst-case execution-time problem - Overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS) 7*, 36:1–36:53.

[100] Witkowski, T., Blanc, N., Kroening, D., and Weissenbacher, G. 2007. Model checking concurrent Linux device drivers. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Atlanta, Georgia, Nov. 2007, R. E. K. Stirewalt, A. Egyed, and B. Fischer, Eds. ACM, New York, NY, 501–504.

[101] Yang, X., Chen, Y., Eide, E., and Regehr, J. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the ACM SIGPLAN 2011 Conference on Programming Language Design and Implementation (PLDI)*, San Jose, CA, June 2011, M. W. Hall and D. A. Padua, Eds. ACM, New York, NY, 283–294.

[102] Yu, D., Hamid, N. A., and Shao, Z. 2003. Building certified libraries for PCC: Dynamic storage allocation. In *Proceedings of the 12th European Symposium on Programming (ESOP)*, Warsaw, Poland, Apr. 2003, P. Degano, Ed. Springer, Berlin, Germany, 363–379.

[103] Yu, D. and Shao, Z. 2004. Verification of safety properties for concurrent assembly code. In *Proceedings of the 9th Proceedings of the International Conference on Functional Programming (ICFP)*, Snow Bird, UT, Sept. 2004, C. Okasaki and K. Fisher, Eds. ACM, New York, NY, 175–188.

[104] Zhao, Y., Huang, Y., He, J., and Liu, S. 2011. Formal model of interrupt program from a probabilistic perspective. In *Proceedings of the 16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, Las Vegas, NV, Apr. 2011, I. Perseil, K. Breitman, and R. Sterritt, Eds. IEEE Computer Society, Washington, DC, 87–94.