Correctness Proofs for Device Drivers in Embedded Systems

Jianjun Duan
School of Computing
University of Utah
jjduan@cs.utah.edu

John Regehr
School of Computing
University of Utah
regehr@cs.utah.edu

Abstract

Computer systems do not exist in isolation: they must interact with the world through I/O devices. Our work, which focuses on constrained embedded systems, provides a framework for verifying device driver software at the machine code level. We created an abstract device model that can be plugged into an existing formal semantics for an instruction set architecture. We have instantiated the abstract model with a model for the serial port for a real embedded processor, and we have verified the full functional correctness of the transmit and receive functions from an open-source driver for this device.

1 Introduction

Formal verification of high-level software is relatively well understood. Interesting recent examples include correctness proofs for a realistic compiler [20], a LISP interpreter [25], and an operating system kernel [18]. Device drivers—routines that directly interact with peripherals—are less well understood, but they are worth verifying since they are part of the trusted computing base for essentially all safety-critical computer systems.

Our goal is to prove full functional correctness of bottom-level device code that directly interacts with hardware devices through I/O registers. Our approach requires substantial manual effort, but the resulting proofs are stronger than those produced in previous work: we guarantee not only that the hardware device is driven in a functionally correct fashion, but also that its timing requirements are met. This approach is too detail-oriented to scale up to large driver stacks that may be thousands of lines; we intend to integrate our manual proofs about bottom-level routines with more automated methods for reasoning about the rest of the driver stack.

Starting with Fox et al.'s semantics for the ARM instruction set architecture [14, 23], which models only the ARM core and memory, we created a framework for plugging in models of hardware peripherals typically found on an ARM-based systems-on-chip (SoC), such as communication ports, timers, and analog-to-digital converters. As a proof-of-concept, we instantiated our abstract device model with a UART (serial port) found on a real ARM processor. We then took a simple open-source driver for this UART, compiled it to ARM object code, and verified the functional correctness of its transmit and receive functions. Our results are all proved in HOL4 [15]. So far, our work is limited to devices and drivers that do not use interrupts or direct memory access (DMA).

2 Timing Requirements

Peripherals are clocked independently of the processor core. Therefore, even on a uniprocessor, device drivers execute in parallel with the devices they manage. One approach to verifying device drivers is to model the hardware as an independent thread, permitting thread verification techniques to be leveraged. However, this approach abstracts away timing information, preventing timing requirements from being verified. Timing constraints are not only important in device drivers, but also embedded systems often have timing constraints as part of their top-level specifications. For example, an automotive brake-by-wire system might be required to begin brake drum actuation within 50 ms of receiving a brake pedal input.

Let's look at a more detailed example: a hypothetical embedded system with two peripherals. One is a device that collects data from the environment at a fixed rate and stores a sample into a hardware FIFO every s cycles. The driver for this sensor copies a sample from the FIFO to a buffer in memory, which costs sd cycles. The other device is a transmitter that sends out of its own hardware FIFO, one sample every t cycles. Its driver copies a sample from the memory buffer into the transmit FIFO, which costs td cycles.

Assume that the drivers are called in a synchronous fashion from a tight loop that repeatedly reads a sample and then transmits a sample. The loop body executes in $sd + td + c$ cycles, where c is the loop overhead.

For a simple system without added synchronization, $t \leq sd + td + c \leq s$ must hold or else samples will be dropped. If $t > sd + td + c$, either the buffer in the transmitter will be overrun or the buffer in memory will be overrun. If $sd + td + c > s$, either the buffer in the sensor will be overrun or the buffer in the memory will be overrun.

To specify and verify properties like this, a rather low-level model of the interaction between devices and their drivers are needed. The rest of this paper describes such a model.

3 Modeling a Processor with Devices

This section describes our system model, which makes it possible to plug devices into an existing model of an ARM processor core. Our device model, like the ARM model, is formalized in HOL4. All theorems listed in this paper have been proved in HOL4.

3.1 Background: Cambridge ARM semantics

Our work is based on Fox et al.'s formal model of the ARMv4 instruction set architecture [14, 23]. It includes banked registers including special purpose registers, exceptions, status flags, co-processors and a data bus. A system model with no co-processors 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 the system state as the input and returns the next state. There are tools to automatically prove theorems about the semantics of individual concrete ARMv4 instructions.

3.2 System model

We model an embedded system using this record:

$$\{\mathit{next} : \mathit{state} \rightarrow \mathit{state}, \mathit{undefed} : \mathit{state} \rightarrow \mathit{bool}\} \quad (1)$$

$\rightarrow$ 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 the instruction, fetching data, computing, updating registers, memory and the state of devices. The *undefed* predicate tests if the state is erroneous. The system enters an erroneous state when the processor core encounters an exceptional condition (our work does not consider the handling of interrupts or processor exceptions), when the processor accesses the memory addresses which are mapped to some device which is not present in the system, or when a device-specific error is encountered (for example, reading a device register that is in an indeterminate state or writing to a read-only device register).

We require that:

$$\forall s.\ \mathit{undefed}\ \ s \Longrightarrow \mathit{undefed}\ \ (\mathit{next}\ \ s) \qquad (2)$$

That is, the erroneous state is sticky and we are not concerned with the system's subsequent behavior. One of our 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*:

$$\begin{aligned} \mathit{step}\ \ n\ \ s\ &= \textbf{if}\ \ n\ = 0\ \ \textbf{then}\ \ s \\ &\qquad \textbf{else}\ \ \mathit{next}\ \ (\mathit{step}\ \ (n-1)\ \ s) \qquad (3) \end{aligned}$$

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, we require the state of the related device contain input streams, which need to contain the information from the future.

3.3 Specifying properties

We use the following construct to specify the properties of a state for the system:

$$\begin{aligned} &\mathit{sys\_pred}\ \ (P,\ I,\ Q) = \\ &\forall s.\ P\ \ s \wedge \neg \mathit{undefed}\ \ s \Longrightarrow \\ &\quad \exists t.\ Q\ \ (\mathit{step}\ \ t\ \ s) \wedge \\ &\qquad (\forall n.\ n \leq t \Longrightarrow \\ &\qquad\quad I\ \ (\mathit{step}\ \ n\ \ s) \wedge \\ &\qquad\quad \neg \mathit{undefed}\ \ (\mathit{step}\ \ n\ \ s)) \qquad (4) \end{aligned}$$

This is a shallow embedding of Hoare logic [13, 16] with *P*, *I*, *Q* as the predicates of precondition, global invariant and postcondition with type $\mathit{state} \rightarrow \mathit{bool}$. 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*. We represent the program as a set of pairs of an instruction and its address. We use *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*.

3.4 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, we design an abstract type to represent a generic peripheral:

$$\begin{aligned}&\{mapped : addr \rightarrow bool,\\ &mappedRead : addr \rightarrow \tau \rightarrow (word * bool * \tau),\\ &mappedWrite : addr \rightarrow word \rightarrow \tau \rightarrow (bool * \tau),\\ &transit : \tau \rightarrow \tau, wellform : \tau \rightarrow bool\} \qquad (5)\end{aligned}$$

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. *mappedRead* and *mappedWrite* 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. *transit* describes the autonomous transition of the device itself without the command from the processor core. *wellform* tells if a state of the device is wellformed.

A concrete device such as a UART 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.

3.5 Extended system model

An embedded SoC is a processor core plus a collection of peripherals. We start with a processor core that is extended with a null device whose *mapped* function returns false for all addresses. We can then build a realistic SoC by adding more devices on top of this bare one, as shown in Figure 1. Device models can be repeatedly composed as long as they fail to share mapped registers (real devices have this property, generally).

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

$$\begin{aligned}&\{regs : regnum \rightarrow word, mem : addr \rightarrow word,\\ &devSt : \tau, \quad undef : bool\}. \qquad (6)\end{aligned}$$

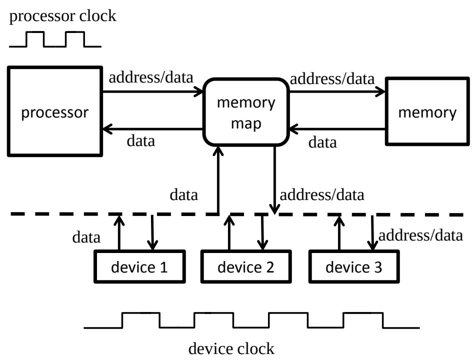

Figure 1: System with devices

Here *regnum* is the type of a register, *regs* represents the register store, which includes the data registers and special purpose registers such the program counter. We use *r0* to indicate register 0, *r14* to indicate register 14, etc. *pc* is used to indicate the program counter. They are all of *regnum* type. *mem* represents the memory. *devSt* represents the state of the devices. The system is in an erroneous state when *undef* is set.

Given a state *s*, *undefed s* = *s.undef*. *next* should implement the execution of the processor core and *transit* for the device in parallel. 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 *mappedRead* or *mappedWrite* and reads data from or writes the data to the specific device register. It may set *undef* based on the results of these operations. At the same time, when running *mappedRead* or *mappedWrite* the device updates its state. The device finally updates its state again with *transit*.

This theorem establishes that adding a new device does not break a system that was previously working:

Theorem 1: If a system *s* does not run into an erroneous state in a *n* steps running a program *p*, it will not run into an erroneous state in *n* steps running *p* with device *d* plugged in.

Proof: It is obvious that *p* does not access the addresses mapped to *d* in these *n* steps. Otherwise it would have run into an erroneous state. So there is no chance for *d* to introduce errors to *p* in these *n* steps, since *s* and *d* are independent of each other in these *n* 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:

Theorem 2: For any system *s* with device dev_a, if *sys_pred (P, I, Q)* holds on it, then it holds for the system with one more device dev_b added, considering only the state components which resembles the state of *s*.

Proof: *sys_pred (P, I, Q)* actually specifies a sequence of transitions of the system. Similar to Theorem 1, in the new system, those components resembling *s* and dev_b are independent of each other in the sequence. So the sequence specified by *sys_pred (P, I, Q)* is still same.

4 A Realistic UART Model

We instantiate the abstract device (5) with a model of the UART0 from an NXP LPC2129 chip [17]. This is a popular embedded processor based on the ARM7TDMI architecture. It targets industrial control applications.

4.1 UART model

Our UART 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 1 summarizes our model's coverage of the UART's register set, where *dlab* stands for divisor latch access bit which controls if registers *DLL* and *DLM* are accessible. It is the 7th bit in the *LCR* register. Our model omits interrupts and modem functionality. The model has an internal buffer with size of one for receiving and transmitting. It does not model line errors or wire encoding since it is assumed that whole characters are transmitted. It does not model the break function.

In our 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 accessed.

4. When data corruption may occur. For example the receiving buffer register is read when its value is indeterminate.

The state of such a UART model is represented with a record:

$$\begin{aligned}\{&RBR : byte,\ \ THR : byte,\ \ SCR : byte,\\ &DLL : byte,\ \ DLM : byte,\\ &dlab : bool,\ \ rdr : bool,\ \ oe : bool,\\ &thre : bool,\ \ temt : bool,\ \ clk : num,\\ &in : num \rightarrow byte\ \ option,\\ &out : num \rightarrow byte\ \ option\};\end{aligned} \tag{7}$$

Here *byte* is the type for 8-bit byte. *num* is the type for the natural number. Note that registers *LCR* and *LSR* are broken down into boolean flags. Access of *FCR* 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 its speed is parameterized relative to the core speed. We are not modeling the exact baud rate. But in a similar fashion we use the 16-bit word value from *DLM* and *DLL* as a slow-down factor *b* unless its value is 0, in which case we set *b* to be 1. For a UART state *ps*, *slowFac ps* returns *b*. The UART only performs meaningful state transition every *b* cycles. To do so, *clk* is incremented for each instruction cycle. But it will be reset to 0 when it reaches *b*. Only at that moment the device performs transmitting and receiving function, updates its registers and shifts its input and output streams. At other moments when $clk < b - 1$, the UART only updates *clk* for book-keeping purpose. However, The memory-mapped access from the processor core however can occur at any *clk* value.

The incoming and outing 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

$$\lambda t.\ in\ \ (t + 1) \tag{8}$$

the new value for *out* is

$$\lambda t.\ \textbf{if}\ \ t = 0\ \ \textbf{then}\ \ d\ \ \textbf{else}\ \ out\ \ (t - 1) \tag{9}$$

where *d* is the character just sent out.

4.2 Describing the property of UART

We describe the property of the UART device in terms of strings extracted from the output queue and input queue. Only non-empty strings are considered. The predicates are defined in Figure 2. Suppose *hd*, *tl* return the first character and the tail of a string respectively, and *TRUE* stands for boolean value true. UART states which are not well-defined are excluded by the *wellform* function. An input stream can be shifted by *cutStrm*. For the transmit function, *outStr s os* describes that *s* is the most recent string in the output stream *os*. And *sentStr s ps* describes that *s* is the most recent string sent out by the processor in the UART state *ps*. For the receive function, *inStr s is* describes that *s* is the string in the input stream *is*. And *inpStr s ps* describes that *s* is the next string to be received by the processor in the UART state *ps*. *inInp m c ps* describes that a character *c* is at most at slot *m* in the input stream of *ps*. *shifted ps2 ps1* is a weak invariant for

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 $\neg dlab$	R	no data received	never	reset data ready flag	none
THR	0	transmit holding when $\neg dlab$	W	never	no room for transmission	none	reset empty transmission queue flags
DLL	0	divisor latch LSB when *dlab*	RW	never	never	none	none
DLM	4	divisor latch MSB when *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 *dlab* flag
LSR	20	line status	R	never	always	reset overrun flag	none
SCR	28	scratch pad	RW	never	never	none	none

Table 1: UART model coverage. R indicates read only registers; W indicates write only. RW indicates no restriction on access. The first four columns are from the LPC2129 manual.

$$\begin{aligned}
&\mathit{wellform}\ \ ps = (\neg ps.temt \lor ps.thre) \land (\neg(ps.clk = \mathit{0}) \lor ps.thre) \land ps.clk < \mathit{slowFac}\ \ ps \\
&\mathit{cutStrm}\ \ n\ \ s = \lambda x.s\ \ (n + x) \\
&\mathit{outStr}\ \ s\ \ os = \exists n.(os\ \ n = \texttt{SOME}\ \ (hd\ \ s)) \land (\forall l.l < n \Longrightarrow (os\ \ l = \texttt{NONE})) \land \mathit{outStr}\ \ (tl\ \ s)(\mathit{cutStrm}\ \ (n + \mathit{1})\ \ os) \\
&\mathit{sentStr}\ \ s\ \ ps = \textbf{if}\ \ \neg ps.thre\ \ \textbf{then}\ \ (ps.THR = h) \land \mathit{outStr}\ \ (tl\ \ s)\ \ ps.out\ \ \textbf{else}\ \ \mathit{outStr}\ \ s\ \ ps.out \\
&\mathit{inStr}\ \ s\ \ is = \exists n.(is\ \ n = \texttt{SOME}\ \ h) \land (\forall l.l < n \Longrightarrow (is\ \ l = \texttt{NONE})) \land \mathit{inStr}\ \ (tl\ \ s)\ \ (\mathit{cutStrm}\ \ (n + \mathit{1})\ \ is) \\
&\mathit{inpStr}\ \ s\ \ ps = \textbf{if}\ \ ps.rdr\ \ \textbf{then}\ \ (ps.RBR = h) \land \mathit{inStr}\ \ (tl\ \ s)\ \ ps.in\ \ \textbf{else}\ \ \mathit{inStr}\ \ s\ \ ps.in \\
&\mathit{inInp}\ \ m\ \ c\ \ ps = ps.rdr \land (c = ps.RBR) \lor \exists n.n < m \land (ps.in\ \ n = \texttt{SOME}\ \ c) \\
&\mathit{shifted}\ \ ps2\ \ ps1 = \exists n.(ps2.in = \mathit{cutStrm}\ \ n\ \ ps1.in) \land \textbf{if}\ \ ps2.rdr\ \ \textbf{then}\ \ \mathit{inInp}\ \ n\ \ ps2.RBR\ \ ps1\ \ \textbf{else}\ \ \mathit{TRUE} \qquad (10)
\end{aligned}$$

Figure 2: Definitions used to describe the property of the UART device

the receive function. It describes that the input queue in *ps2* results from the one in *ps1* after some time.

5 Correctness of a UART Driver

We 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. We 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. We did this because "bx" is not modeled in our current ARM tool which does not support the THUMB mode. We proved full correctness for three functions which interact with device registers: the *putch* function transmits a character, the *getch* function which attempts to read a character from *RBR*, and the *getchW* function which performs a blocking read from *RBR*. The code is shown in Figure 3.

5.1 UART model soundness

Our correctness claim is based on the correctness claims of other components. We 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 we depend on the fact that the abstract device model attached to the ARM model is sound as shown in Section 3.5. The soundness of the UART model is proved in the process. For example, We have proved the following properties regarding the transmitting function, among others:

Theorem 3: No character will be appended to the output under any following conditions:

1. no memory-mapped read or write occurs,
2. a read occurs,
3. 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 scenario in which a character is appended to the output is when the *THR* register is written with *thre* set.

```
<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: The ARM assembly code for *putch*, *getch* and *getchW*

5.2 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. They are a useful part of the safety properties from the correctness specification, and also they 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, we could know what addresses or registers are accessed in an ARM instruction when it is decoded. Here we use a different approach by examining the change of content in the memory and registers. Since the access which could cause side effects is limited to access of memory-mapped device registers, of which we already take care, this approach serves our purpose well. Function *sepMem accSet s1 s2* just does this. It checks two system states *s1* and *s2* to see if any address not in the set *addrSet* have the same content. *sepReg regSet s1 s2* does the same thing for the registers across two states. These two predicates are rather naive, an embedding of separation logic here would be nice.

Control flow integrity specifies that only some certain sequences of PC values can occur in the execution. For example, when *putch* is busy waiting, it just 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 we did not include the stepwise specification of the control flow.

5.3 Correctness of the UART driver

We proved the full correctness theorems of 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 *rdr* is set. Otherwise it returns *0xff*. *getchW* first waits for *rdr* being set. It will then copy the byte from *RBR* to register *r0*. Note that if *getchW* is used to receive a string, character may be dropped if the UART 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. And the basic safety property states that memory safety is observed, the operating configuration of the UART device is not changed in terms of its speed (described by the slow-down factor) and the controlling bit *dlab*, and the system does not run into any erroneous state. The following three theorems states the correctness of the three functions.

Theorem 4: *putch* will successfully appended the character from $r0$ to the string already sent out in the output queue. The basic safety and liveness properties hold in the process.

Theorem 5: *getch* will successfully read a character from the input queue or return *0xff*. The basic safety and liveness properties hold in the process.

Theorem 6: If there is a string in the input queue, and the UART is slow enough, function *getchW* will successfully read the next character from the input stream. In the process, no overrun error occurs to the UART, and the basic safety and liveness properties hold.

The correctness of *getchW* depends on the speed of the UART device relative to the ARM core, and the latency caused by the driver code. The driver code must be efficient enough and the UART must be slow enough so that no buffer overrun error can occur. Our approach allows such constraint to be expressed, while the previous work is rather awkward at this [1].

The tight timing properties in Theorem 6 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 slow-down factor of the UART and the latency introduced in *getchW*, which is 9 instruction cycles. This guarantees that *oe* is not set when *getchW* is entered

	putch	getch	getchW
A			$\neg(\mathit{length}\ \ \mathit{str} = \mathit{0}) \wedge$ $\mathit{9} + \mathit{delay} < (\mathit{slowFac}\ \ \mathit{s0})$
P	$\lambda s.\ (\mathit{code}\ \ p\ \ s) \wedge$ $(s.\mathit{regs}\ \ \mathit{pc} = \mathit{0x28c}) \wedge$ $(s.\mathit{regs}\ \ \mathit{r14} = \mathit{reAddr}) \wedge$ $(\mathit{wellform}\ \ s.\mathit{devSt}) \wedge$ $\neg s.\mathit{devSt}.\mathit{dlab} \wedge$ $(\mathit{LSB}\ \ (s.\mathit{regs}\ \ \mathit{r0}) = c) \wedge$ $(\mathit{sentStr}\ \ \mathit{str}\ \ s.\mathit{devSt})$	$\lambda s.\ (\mathit{code}\ \ p\ \ s) \wedge$ $(s.\mathit{regs}\ \ \mathit{pc} = \mathit{0x314}) \wedge$ $(s.\mathit{regs}\ \ \mathit{r14} = \mathit{reAddr}) \wedge$ $(\mathit{wellform}\ \ s.\mathit{devSt}) \wedge$ $\neg s.\mathit{devSt}.\mathit{dlab} \wedge$ $\neg s.\mathit{devSt}.\mathit{rdr}$	$\lambda s.\ (\mathit{code}\ \ p\ \ s) \wedge$ $(s.\mathit{regs}\ \ \mathit{pc} = \mathit{0x334}) \wedge$ $(s.\mathit{regs}\ \ \mathit{r14} = \mathit{reAddr}) \wedge$ $(\mathit{wellform}\ \ s.\mathit{devSt}) \wedge$ $\neg s.\mathit{devSt}.\mathit{dlab} \wedge$ $(\mathit{inpStr}\ \ \mathit{str}\ \ s.\mathit{devSt}) \wedge$ $\neg s.\mathit{devSt}.\mathit{oe}$
I	$\lambda s.\ (\mathit{sepMem}\ \ \mathit{lpcMapped}\ \ \mathit{s0}\ \ s) \wedge$ $(\mathit{sepReg}\ \ \mathit{putchReg}\ \ \mathit{s0}\ \ s) \wedge$ $\neg s.\mathit{devSt}.\mathit{dlab} \wedge$ $(\mathit{wellform}\ \ s.\mathit{devSt}) \wedge$ $(\mathit{slowFac}\ \ s = \mathit{slowFac}\ \ \mathit{s0})$	$\lambda s.\ (\mathit{sepMem}\ \ \mathit{lpcMapped}\ \ \mathit{s0}\ \ s) \wedge$ $(\mathit{sepReg}\ \ \mathit{getchReg}\ \ \mathit{s0}\ \ s) \wedge$ $\neg s.\mathit{devSt}.\mathit{dlab} \wedge$ $(\mathit{wellform}\ \ s.\mathit{devSt}) \wedge$ $(\mathit{slowFac}\ \ s = \mathit{slowFac}\ \ \mathit{s0})$	$\lambda s.\ (\mathit{sepMem}\ \ \mathit{lpcMapped}\ \ \mathit{s0}\ \ s) \wedge$ $(\mathit{sepReg}\ \ \mathit{getchWReg}\ \ \mathit{s0}\ \ s) \wedge$ $\neg s.\mathit{devSt}.\mathit{dlab} \wedge \neg s.\mathit{devSt}.\mathit{oe} \wedge$ $(\mathit{wellform}\ \ s.\mathit{devSt}) \wedge$ $(\mathit{slowFac}\ \ s = \mathit{slowFac}\ \ \mathit{s0})$
Q	$\lambda s.\ (s.\mathit{regs}\ \ \mathit{pc} = \mathit{reAddr}) \wedge$ $(\mathit{sentStr}\ \ (c :: \mathit{str})\ \ s.\mathit{devSt})$	$\lambda s.\ (s.\mathit{regs}\ \ \mathit{pc} = \mathit{reAddr}) \wedge$ $((\mathit{LSB}\ \ (s.\mathit{regs}\ \ \mathit{r0}) = \mathit{0xff}) \vee$ $\exists m.\ \mathit{inInput}\ \ m\ \ (s.\mathit{regs}\ \ \mathit{r0})$ $s.\mathit{devSt})$	$\lambda s.\ (s.\mathit{regs}\ \ \mathit{pc} = \mathit{reAddr}) \wedge$ $(\mathit{LSB}\ \ (s.\mathit{regs}\ \ \mathit{r0}) = \mathit{el}\ \ \mathit{0}\ \ \mathit{str}) \wedge$ $(\mathit{inpStr}\ \ (\mathit{tl}\ \ \mathit{str})\ \ s.\mathit{devSt}) \wedge$ $\neg s.\mathit{devSt}.\mathit{rdr} \wedge$ $s.\mathit{devSt}.\mathit{clk} + \mathit{delay} + \mathit{1} < (\mathit{slowFac}\ \ s)$

Table 2: The assumption A, precondition P, invariant I and postcondition Q for UART driver functions *putch*, *getch* and *getchW*. They are intended to be used in $A \Longrightarrow \mathit{sys\_pred}\ \ (P, I, Q)$. *s0* indicates the initial state at the entry of the respective functions. *p* indicates the respective function body. *reAddr* indicates the return address.

next time, thus the precondition of *getchW* is met.

5.4 Proof method

All the theorems about the execution we proved are in the form of *sys_pred (P, I, Q)*. The details are listed in Table 2. We use *LSB* to extract the least significant byte from a register, which is 32 bits wide in the ARM model that we use. *c::str* appends a character *c* to the head of a string *str*. The modifiable register sets are defined in *putchReg* = *getchWReg* = $\{$*R0, R2, 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 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 is not synchronized with the ARM core at a known rate. In the proof, we provide the witness for the existentially qualified *t*, then use induction on time *n* as in definition (4).

Use *putch* as an example. We break the execution sequence into three parts. With each part we prove a correctness lemma in the form of *sys_pred (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 UART. The length of this waiting depends on the speed of the UART and the UART 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 do not need to work at the level of device details. Sophisticated program logic [11, 23, 24, 32] may be needed to deal with scaling and more complicated control flow. The state of device can be trivially plugged in the proof based on Section 3.5. Our theorems already imply the calling convention. It should not be difficult to translate them into appropriate format and integrate them into high level proof.

6 Related Work

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 OS [6].

Device driver verification: In previous work on verifying functional correctness of device drivers [1, 2, 22], parallelism is modeled as concurrency between the driver code and device transitions, and the cases of interleaving were reduced by considering the net effect of the interleaving on the state of the system. This approach allows one to take advantage of methodologies used in verifying concurrent programs [5, 10, 12, 19, 26]. However, it has difficulty dealing with some timing constraints on the driver code [1]. In contrast, we modeled the speed of the serial 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.

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 [7, 21, 28, 31, 33]. 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.

Property checking: Using model checking to verify temporal properties of device drivers for commodity operating systems is well studied [3, 4, 27]. Most of these works are at the source code level and focus on the interface between the drivers and the kernel, as opposed to focusing on correct interaction with the device. The main goal of most of these works are to keep drivers from crashing or hanging the OS. Source-level model checking can be difficult to use in the context of embedded systems [30]. Model checking of embedded C code [9] and assembly code [8, 29] 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.

7 Conclusion and Future Work

Our goal is to prove full correctness, including timing properties, of device drivers for an embedded system in terms of a model of a CPU plus its peripheral devices. We introduced an abstract device model that can be integrated with a formal model of a processor core and we instantiated it with a realistic model of the UART from a commonly-used ARM processor. We then proved full correctness of the transmit and receive functions from an open-source driver for that device.

Our work is intended to provide a platform for verifying embedded systems in a modular way with regard to its hardware devices. It allows us to prove the correctness of the driver for one device at a time, and claim validity for a system containing multiple devices. For the existing proof about ARM code in HOL4 which does not consider devices, this allows the support for devices being added without repeating most of the proof.

We have three things planned for the future. The first is to finish the proof of the full correctness of the UART driver for the receiving and transmitting function and integrate it with an existing proof. The second one is to design a program logic on top of the existing work to support the devices in a modular way, which will make the reasoning of large programs more scalable. The logic should have separation logic support for the main memory. The third one is to design a framework to support the reasoning about drivers that handle interrupts.

References

[1] Alkassar, E., Hillebrand, M., Knapp, S., Rusev, R., and Tverdyshev, S. Formal device and programming model for a serial interface. In *Proc. of the 4th Intl. Verification Workshop (VERIFY)* (Bremen, Germany, July 2007), pp. 4–20.

[2] Alkassar, E., and Hillebrand, M. A. Formal functional verification of device drivers. In *Proc. of the 2nd Intl. Conf. on Verified Software: Theories, Tools, Experiments (VSTTE)* (Toronto, Canada, Oct. 2008), pp. 225–239.

[3] Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S. K., and Ustuner, A. Thorough static analysis of device drivers. In *Proc. of the 2006 EuroSys Conf.* (Leuven, Belgium, Apr. 2006), pp. 73–85.

[4] Ball, T., and Rajamani, S. K. Automatically validating temporal safety properties of interfaces. In *Proc. of the 8th Intl. SPIN Workshop on Model Checking Software (SPIN)* (Toronto, Canada, May 2001), pp. 103–122.

[5] Chaki, S., Clarke, E., Groce, A., Jha, S., and Veith, H. Modular verification of software components in C. In *Proc. of the 22nd Intl. Conf. on Software Engineering (ICSE)* (Portland, OR, May 2003), pp. 385–395.

[6] Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. R. An empirical study of operating system errors. In *Proc. of the 18th ACM Symp. on Operating Systems Principles (SOSP)* (Banff, Canada, Oct. 2001), pp. 73–88.

[7] Conway, C. L., and Edwards, S. A. NDL: A domain-specific language for device drivers. In *Proc. of the 2004 Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)* (Washington, DC, June 2004).

[8] Fehnker, A., Huuck, R., Rauch, F., and Seefried, S. Some assembly required - program analysis of embedded system code. In *Proc. of the 8th Intl. Working Conf. on Source Code Analysis and Manipulation (SCAM)* (Beijing, China, Sept. 2008), pp. 15–24.

[9] Fehnker, A., Huuck, R., Schlich, B., and Tapp, M. Automatic bug detection in microcontroller software by static program analysis. In *Proc. of the 35th Conf. on Current Trends in Theory and Practice of Computer Science (SOFSEM)* (Spindleruv Mlýn, Czech Republic, Jan. 2009), pp. 267–278.

[10] Feng, X., and Shao, Z. Modular verification of concurrent assembly code with dynamic thread creation and termination. In *Proc. of the 10th ACM SIGPLAN Intl. Conf. on Functional Programming (ICFP)* (Tallinn, Estonia, Sept. 2005), pp. 254–267.

[11] Feng, X., Shao, Z., Guo, Y., and Dong, Y. Certifying low-level programs with hardware interrupts and preemptive threads. *J. Automatic Reasoning 42*, 2-4 (Apr. 2009), 301–347.

[12] Flanagan, C., and Qadeer, S. Thread-Modular model checking. In *Proc. of the 10th Intl. SPIN Workshop on Model Checking Software (SPIN)* (Portland, OR, May 2003), pp. 213–224.

[13] Floyd, R. W. Assigning meanings to programs. In *Proc. of Symp. in Applied Mathematics* (New York City, NY, Apr. 1966), vol. 19, pp. 19–32. Mathematical Aspects of Computer Science.

[14] Fox, A. Formal specification and verification of ARM6. In *Proc. of the 16th Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOLs)* (Rome, Italy, Sept. 2003), pp. 25–40.

[15] Gordon, M. J. C. Introduction to the HOL system. In *Proc. of the 1991 Intl. Workshop on the HOL Theorem Proving System and its Applications (TPHOLs)* (Davis, CA, Aug. 1991), pp. 2–3.

[16] Hoare, C. A. R. An axiomatic basis for computer programming. *Communications ACM 12*, 10 (Oct. 1969), 576–583.

[17] Keil. NXP LPC2129. `http://www.keil.com/dd/chip/3648.htm`.

[18] 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. seL4: formal verification of an OS kernel. In *Proc. of the 22nd ACM Symp. on Operating Systems Principles (SOSP)* (Big Sky, MT, Oct. 2009), pp. 207–220.

[19] Lamport, L. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems 16*, 3 (May 1994), 872–923.

[20] Leroy, X. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. of the 33rd Symp. on Principles of Programming Languages (POPL)* (Charleston, SC, Jan. 2006), pp. 42–54.

[21] Mérillon, F., and Muller, G. Dealing with hardware in embedded software: A general framework based on the Devil language. In *Proc. 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), pp. 121–127.

[22] Monniaux, D. Verification of device drivers and intelligent controllers: a case study. In *Proc. of the 7th Intl. Conf. on Embedded Software (EMSOFT)* (Salzburg, Austria, Sept.–Oct. 2007), pp. 30–36.

[23] Myreen, M. O., Fox, A. C. J., and Gordon, M. J. C. Hoare logic for ARM machine code. In *Proc. of the 2007 Symp. on Fundamentals of Software Engineering (FSEN)* (Tehran, Iran, Apr. 2007), pp. 272–286.

[24] Myreen, M. O., and Gordon, M. J. C. Hoare logic for realistically modelled machine code. In *Proc. of the 13th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (Braga, Portugal, Mar.–Apr. 2007), pp. 568–582.

[25] Myreen, M. O., and Gordon, M. J. C. Verified LISP implementations on ARM, x86 and PowerPC. In *Proc. of the 22nd Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOLs)* (Munich, Germany, Aug. 2009), pp. 359–374.

[26] O'Hearn, P. W. Resources, concurrency, and local reasoning. *Theoretical Computer Science 375*, 1-3 (May 2007), 271–307.

[27] Post, H., and Küchlin, W. Integrated static analysis for Linux device driver verification. In *Proc. of the 6th Intl. Conf. on Integrated Formal Methods (IFM)* (Oxford, UK, July 2007), pp. 518–537.

[28] Ryzhyk, L., Chubb, P., Kuz, I., Sueur, E. L., and Heiser, G. Automatic device driver synthesis with Termite. In *Proc. of the 22nd ACM Symp. on Operating Systems Principles (SOSP)* (Big Sky, MT, Oct. 2009), pp. 73–86.

[29] Schlich, B. Model checking of software for microcontrollers. *ACM Transactions on Embedded Computing Systems (TECS) 9*, 4 (Mar. 2010).

[30] Schlich, B., and Kowalewski, S. Model checking C source code for embedded systems. *Intl. J. Software Tools for Technology Transfer 11*, 3 (Mar. 2009), 187–202.

[31] Sun, J., Yuan, W., Kallahalla, M., and Islam, N. HAIL: a language for easy and correct device access. In *Proc. of the 2005 Intl. Conf. on Embedded Software (EMSOFT)* (Jersey City, NJ, Sept. 2005), pp. 1–9.

[32] Tan, G., and Appel, A. W. A compositional logic for control flow. In *Proc. of the 7th Intl. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI)* (Charleston, SC, Jan. 2006), pp. 80–94.

[33] Wang, S., and Malik, S. Synthesizing operating system based device drivers in embedded systems. In *Proc. of the 1st IEEE/ACM/IFIP Intl. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)* (Newport Beach, CA, Oct. 2003), pp. 37–44.