Formally Verifying FreeRTOS' Interprocess Communication Mechanism

Nathan Chong
Amazon Web Services

Bart Jacobs
imec-DistriNet, KU Leuven

ABSTRACT

FreeRTOS is a real-time kernel and set of libraries for Internet of Things (IoT) applications. The FreeRTOS kernel provides a portable abstraction layer, task scheduling and interprocess communication (IPC) mechanisms. The main IPC mechanism in FreeRTOS is a concurrent queue: a circular buffer data structure that tasks and interrupt service routines use to exchange messages. As a fundamental building block for larger applications, the correctness of the queue implementation is vital. We have formally verified the memory safety, thread safety and functional correctness of this queue implementation using deductive verification. Our proofs are publicly available and give machine-checkable assurances of correctness that would be infeasible to obtain through testing alone.

1 INTRODUCTION

FreeRTOS is a market-leading real-time kernel and set of IoT libraries that enables developers to easily and securely build, deploy and manage IoT applications. The breadth and reach of FreeRTOS applications across multiple industry sectors means that security and correctness are of prime importance.

The reliability of FreeRTOS is ensured through continual investment in security and software quality. This includes strict MISRA coding standards, linting and static checking, code coverage, code reviews, testing in continuous integration and extensive device platform stress testing.[1]

Increasingly for FreeRTOS, this also includes the complementary use of *formal verification* techniques, which enable even higher assurances of correctness through machine-checkable proofs. A key benefit of these techniques is the potential to achieve levels of assurance that we could not obtain through testing.

In the context of formal verification, a proof shows that a program behaves correctly with respect to a logical specification. Logical specifications capture the intent of the developer more precisely than documentation because proofs can link these specifications to the implementation of the program. In particular, formal verification techniques allow us to reason rigorously about *all* of a code's behaviors: what it can do, what it must do and, just as importantly, what it can never do. Advances in underlying techniques and tools have made formal verification of software more tractable and academic research continues to push the boundary in many exciting directions including verified compilers, operating systems and distributed systems [19]. However, formal verification of software in industry is not mainstream due to the costs of *proof engineering*: the practice of developing and maintaining proofs. A principled approach is to apply formal verification to components of a codebase that merit the costs and benefits.

One such component in the FreeRTOS kernel is the concurrent queue implementation for interprocess communication (IPC). Due to preemptive scheduling, the queue must be correct in the presence of multiple tasks concurrently calling queue API operations. For example, consider two tasks that concurrently attempt to send two separate messages to the queue. It is imperative that the integrity of each message be maintained by ensuring that the queue is updated consistently regardless of the state of the queue and in the presence of other tasks. Careful design and thorough testing gives assurance that this is the case, but formal verification can provide higher assurance by considering all such scenarios, particularly subtle concurrent scenarios that would be difficult or burdensome to trigger reliably through testing.

We have verified the FreeRTOS queue implementation in the presence of this concurrent behavior. Informally, the proofs show that the FreeRTOS queue implementation is memory safe (i.e., does not access invalid memory or dereference `NULL` pointers), thread safe (i.e., properly synchronizes accesses to shared memory to avoid data races) and functionally correct (i.e., behaves like a queue). Furthermore, these properties hold regardless of the number of tasks accessing the queue or the thread schedule. Our proofs are publicly available and run as part of FreeRTOS continuous integration since V10.4.3, in order to help ensure their continued maintenance.[2]

The verification was performed using the VeriFast deductive verifier [14], taking three person months to develop proofs for 14 API functions and 6 internal functions (approximately 700 lines of code) of the queue implementation. The verification time for all proofs takes less than 5 seconds on a commodity laptop.[3] Table 1 gives a per-function breakdown of the proof annotation overhead, which ranges between 0.3–2×. Proof annotations are statements added to the code to aid verification and we give an example of a function with proof annotations in §4. We note that the landscape of formal verification is broad and there are more automated techniques such as model checking [6, Chapter 2] that do not require proof annotations. Indeed, Amazon has successfully applied model checking to FreeRTOS[4] and other low-level C-based systems [5, 8], however, these techniques do not typically scale to reasoning about concurrent code. The contributions of this work are:

- Formal verification of the FreeRTOS concurrent queue implementation, the main IPC mechanism of FreeRTOS (§4)
- Four findings in the queue implementation that have been reported to and addressed by the FreeRTOS developers (§5)

[1] "It is worth noting that the thoroughness of these [FreeRTOS] tests have been responsible for finding bugs in silicon on multiple occasions", https://www.freertos.org/FreeRTOS-Coding-Standard-and-Style-Guide.html

[2] https://github.com/FreeRTOS/FreeRTOS/tree/master/FreeRTOS/Test/VeriFast

[3] Intel 2.5GHz i7, 16GB RAM MacBook Pro with VeriFast 19.12

[4] https://www.freertos.org/2020/02/ensuring-the-memory-safety-of-freertos-part-1.html

Table 1: Per-function proof LOA (lines of annotation) to verify LOC (lines of code) of the concurrent queue implementation of FreeRTOS

	LOC	LOA	Annotation Overhead
API FUNCTIONS:			
uxQueueMessagesWaiting	9	4	44%
uxQueueMessagesWaitingFromISR	9	2	22%
uxQueueSpacesAvailable	13	5	38%
vQueueDelete	26	8	31%
xQueueGenericCreate	62	21	34%
xQueueGenericReset	33	17	52%
xQueueGenericSend	93	29	31%
xQueueGenericSendFromISR	74	22	30%
xQueueIsQueueEmptyFromISR	14	3	21%
xQueueIsQueueFullFromISR	14	3	21%
xQueuePeek	76	22	29%
xQueuePeekFromISR	28	8	29%
xQueueReceive	74	28	38%
xQueueReceiveFromISR	42	14	33%
INTERNAL FUNCTIONS:			
prvCopyDataFromQueue	18	31	172%
prvCopyDataToQueue	46	52	113%
prvIsQueueEmpty	12	4	33%
prvIsQueueFull	12	4	33%
prvLockQueue	18	8	44%
prvUnlockQueue	51	10	20%
SHARED PROOFS:			
15 definitions and 42 lemmas		671	
TOTAL	724	966	133%

1.1 Proof Assumptions

As is the case for all verified software, our proofs are subject to assumptions which must be carefully reviewed to ensure they are reasonable. Generally our proofs assume well-behaved applications, the correctness of underlying primitives and system behavior.

- The specification is a *contract*: if the application adheres to the queue API specification then in return the proofs guarantee correctness properties. Proofs show that an implementation is valid with respect to its specification. Hence, a badly-behaved application can invalidate our proofs if the specification requirements are not followed. For example, an application that reads or writes to the queue storage directly, without using the queue API, invalidates thread safety. The specification forbids this behavior but we cannot, in general, enforce this behavior since we do not verify application code.
- We assume the memory safety, thread safety and functional correctness for primitives used by the queue implementation for memory allocation and task scheduling.[5] We provide a specification for each primitive but we do not verify their implementation. We note that in some cases there is no single implementation—they are an application-level choice or specialized by each device platform.
- We assume a system property regarding the isolation guaranteed between tasks and interrupt service routines (ISRs). Specifically, we assume that the macro `taskENTER_CRITICAL` and its equivalent for ISRs, which FreeRTOS implements on a per-platform basis as interrupt masking gives strong isolation [3]. An informal example of strong isolation is:

Initially `x == 0`	

`// Task 1`
`taskENTER_CRITICAL()`
`x = 1;`
`x = 2;`
`taskEXIT_CRITICAL()`
	`// Task 2`
	`r = x;`

Assert `r == 0` or `r == 1`	

That is, strong isolation of Task 1's critical section means Task 2 must never see the intermediate state `x == 1`. We discuss this assumption further in Section 3.2 when we describe the concurrency mechanisms used by the queue.

Regarding the trusted computing base of our proofs: we rely on VeriFast, the C Compiler and the underlying hardware. Issues in any of these components could effect the soundness of our proofs. Regarding VeriFast, a core subset of the underlying technique has been formalized [21], which increases our confidence in the tool. Trusting the C compiler and underlying hardware is necessary because our proofs are performed at the level of the C implementation code.

2 RELATED WORK

FreeRTOS has been the subject of several formal verification efforts [4, 5, 9–11, 15, 18, 20]. Cheng et al. give a good overview of these efforts up to 2015. These can broadly be divided into two types: specification-level and implementation-level. At the specification-level, the functionality of portions of FreeRTOS are *modeled* in an abstract specification language such as B method [9] or Z notation [4]. At the implementation-level, the code of FreeRTOS is analyzed more directly [5, 11, 15, 20]. The work of Divakaran et al. bridges these two styles and presents a proof linking an abstract specification in Z notation to the implementation-level code of the FreeRTOS task scheduler using the VCC deductive verifier [7].

The work of Ferreira et al. is most closely related to ours. Similar to us, they use deductive verification based on separation logic (§3.3) to reason about the FreeRTOS task scheduler and its underlying list data structure. The annotation overhead reported by their work (ranging between 0.13–2× [11, Table 1]) compares similarly to ours (Table 1). However, their work relies on a hand-translation of C code into an intermediate imperative language, whereas we reason directly on the source. Our work differs from prior implementation-level verification efforts in two main ways. Firstly, we verify concurrent code whereas prior efforts have focused on sequential proofs or assumed atomicity at the API-level. Secondly, the proofs in this paper are maintained in-sync with FreeRTOS development by being

[5] Specifically: `pvPortMalloc`, `vPortFree`, `memcpy`, `vListInitialise`, `xTaskRemoveFromEventList`, `vTaskMissedYield`, `xTaskCheckForTimeOut`, `vTaskInternalSetTimeOutState` and `vTaskPlaceOnEventList`

part of continuous integration. This is also the case for the model checking proofs discussed by Chong et al..

3 BACKGROUND

We briefly overview the execution model of FreeRTOS before moving onto the queue: its layout, sequential behavior and concurrent behavior. These topics are also covered in the FreeRTOS documentation [1, 2].

3.1 FreeRTOS Execution Model

An application using FreeRTOS is partitioned into tasks and interrupt service routines (ISRs). A task is a thread of computation with its own stack under the control of the FreeRTOS task scheduler. An ISR is a procedure registered to an interrupt. ISRs are run in an event-driven manner under the control of the interrupt controller (e.g., the NVIC on Arm Cortex-M systems). Every task has a priority. Higher-priority tasks can preempt lower-priority tasks when the task scheduler is configured to use preemptive scheduling. Interrupts always preempt tasks and nest if a higher-priority interrupt occurs. Tasks can never preempt ISRs. Every FreeRTOS application contains a lowest-priority idle task and context-switching ISR.

3.2 FreeRTOS Queue Implementation

The kernel queue implementation is given in `queue.c`.[6] The file is approximately 2K LOC but only 700 LOC is used to implement the queue. The remainder builds synchronization objects, such as semaphores and mutexes, out of the queue implementation. Queues are created and used by an application through a queue API provided by the kernel. The main API functions are given in Table 1.

Layout. Figure 1 shows the layout of a queue of N elements of M bytes in memory (i.e., the queue can store at most N fixed-size messages). In the queue struct, N and M are stored in the fields `uxLength` and `uxItemSize`. All queues have $0 < N$ and $0 < M$. The pointers `pcHead` and `pcTail` delimit the buffer: `pcHead` (respectively, `pcTail`) points-to the first (respectively, one-byte after the last byte) of the storage buffer. These pointers are fixed. The pointers `pcReadFrom` and `pcWriteTo` are the front and back of K *valid* elements in the queue. In the queue struct, K is stored in the field `uxMessagesWaiting`. These pointers can point at any element boundary in the buffer with the valid elements circularly wrapping around if `pcWriteTo` < `pcReadFrom`. Initially, at queue creation and reset, `pcReadFrom` (respectively, `pcWriteTo`) points-to the last (respectively, first) element of the buffer corresponding to $K = 0$ valid elements. We view the fields of the queue struct and the buffer itself as *resources* that a task or ISR can access. The figure omits queue resources for task blocking behavior—task wait lists and queue locks—which we defer to the section on *Concurrent Behavior*, below.

Sequential Behavior. A task or ISR with a reference to the queue can send a message (place an element into the queue) or receive a message (take an element from the queue) by calling the queue API. Figure 2 shows the state of a queue with $N = 4$ elements after successive send and receive operations, which we assume to be sequential, starting with the reset state `S0`. A send operation copies the message into the element pointed-to by the back of the queue, `pcWriteTo` (denoted `W`), and then increments the pointer by one element modulo N to account for wraparound behavior (e.g., `S0` ⟶ `S1`). A receive operation increments the front of the queue, `pcReadFrom` (denoted `R`), by one element modulo N to account for wraparound behavior and copies the element pointed-to by the new front of the queue to the result (e.g., `S2` ⟶ `S3`). The queue API also supports sending messages to the front of the queue or overwriting the contents of an element (in the case when $N = 1$).

Concurrent Behavior. The queue API supports multiple tasks and ISRs sending and receiving into the queue at the same time and allows non-blocking and blocking behavior for tasks. Under non-blocking behavior, a send (respectively receive) API call returns with an error if the queue is full (respectively empty). Whereas, under blocking behavior, the calling task will wait up to a task-specified timeout for the queue state to change (i.e., the queue to become not-full in the case of a send or non-empty for a receive) to enable the call to succeed. If the timeout is reached then an error is returned. In the queue, blocking behavior is implemented by appending the task control block (TCB) of the calling task to a task wait list `xTasksWaitingToSend` for send (respectively `xTasksWaitingToReceive` for receive) stored in the queue struct. ISRs cannot call blocking API queue functions because an interrupt must never block as this would deadlock the system since context-switching is performed using a dedicated context-switching ISR.

Since kernel synchronization mechanisms, such as semaphores and mutexes, are built on top of the queue implementation, the queue itself must use lower-level mechanisms. The queue implementation uses three kinds of concurrency control:

- **Interrupt masking.** FreeRTOS provides macros to mask interrupts which allow the system to ignore interrupts below a certain priority until the mask is cleared.[7] On a uniprocessor system this mechanism provides strongly isolated *critical sections* between tasks and ISRs. A task in such a critical section cannot be preempted by interrupts or other tasks (even higher-priority tasks) since context-switching is performed by an ISR. Similarly, an ISR in such a critical section cannot be preempted by interrupts (and tasks can never preempt interrupts (§3.1)). A task or ISR using this mechanism has exclusive access to almost all queue resources.
- **Scheduler suspension.** Suspending the task scheduler disables context-switching and provides critical sections between tasks.[8] Note that this mechanism does not provide isolation between tasks and ISRs. In the case of the queue, there are no task-only resources so this mechanism, by itself, does not give exclusive access to queue resources.
- **Lock variables.** Both interrupt masking and task scheduling suspension are generic mechanisms. Specific to the queue implementation are two 'lock' variables `cTxLock` and `cRxLock` of type `int8_t`. The lock `cTxLock` (respectively, `cRxLock`) protects the task wait list `xTasksWaitingToReceive` (respectively, `xTasksWaitingToSend`). The value −1 denotes `unlocked` and values `0...127` denote `locked`; the values

[6] https://github.com/FreeRTOS/FreeRTOS-Kernel/blob/main/queue.c

[7] Tasks use `taskENTER_CRITICAL` and `taskEXIT_CRITICAL` and ISRs use `portSET_INTERRUPT_MASK_FROM_ISR` and `portCLEAR_INTERRUPT_MASK_FROM_ISR`

[8] Using `vTaskSuspendAll` and `vTaskResumeAll`

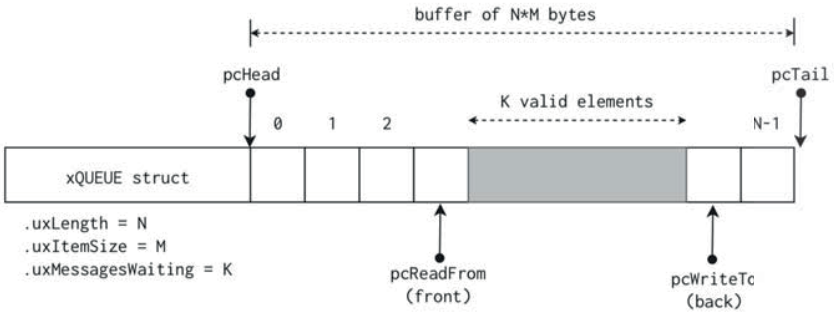

Figure 1: Queue layout in memory

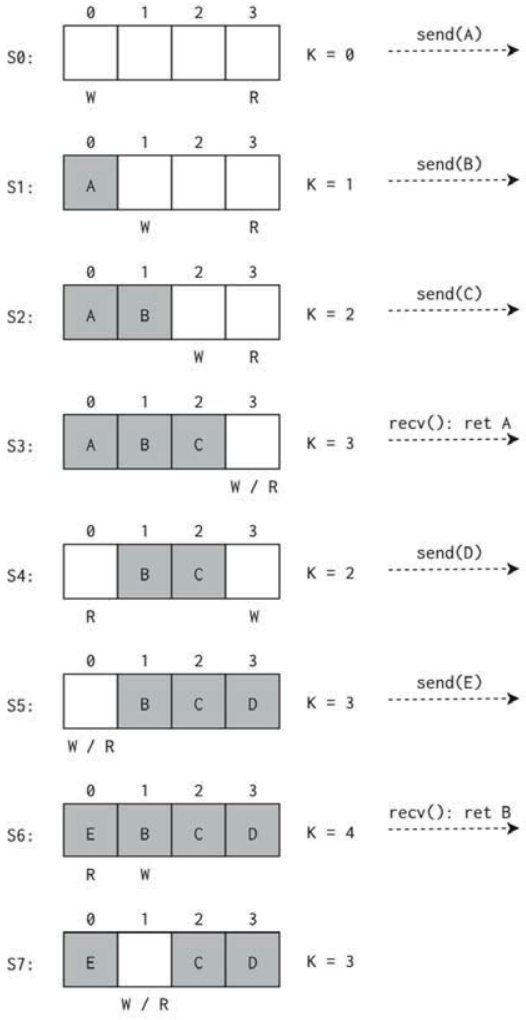

Figure 2: Sequential behavior of a queue with $N = 4$. We use `R` and `W` to denote the front and back of the queue (i.e., `pcReadFrom` and `pcWriteTo`).

$-128 \ldots -2$ are invalid. A non-zero value i for `cTxLock` (respectively, `cRxLock`) indicates that i ISRs have sent to (respectively, received from) the queue in the interval between the task locking and unlocking.

The queue implementation uses interrupt masking for atomic updates to the queue (i.e., copying data to and from the queue), but scheduler suspension with locking for task blocking behavior (accessing the task wait lists). Although interrupt masking is sufficient for preserving isolation in all cases, FreeRTOS uses these two mechanisms in conjunction for performance reasons to avoid masking interrupts for long periods of time. If a task is in a critical section with *both* task scheduling suspended and a lock then it has exclusive access to the task wait list and can safely add its TCB into the list to block. A task in such a critical section cannot be preempted by other tasks due to task scheduler suspension but may be interrupted. In this case, if the corresponding ISR is in a critical section with interrupts masked and the lock is held (i.e., by the interrupted task) then the ISR must not access the task wait list, but does have exclusive access to all other queue resources; otherwise, the lock is free and the ISR can safely access the task wait list in order to wake up any tasks that may now be unblocked. Table 2 summarizes the interaction of these mechanisms with respect to the queue resources.

Table 2: Summary of ownership of queue resources using concurrency control mechanisms Irq (interrupt masking), Sched (scheduler suspension) and Lock (queue locks)

	Irq	Sched	Lock	Resource
Task	0	0	0	None
	0	0	1	Invalid
	0	1	0	None
	0	1	1	Wait lists
	1	0	0	All excluding wait lists
	1	0	1	Invalid
	1	1	0	All excluding wait lists
	1	1	1	All including wait lists
ISR	0	X	X	None
	1	X	0	All including wait lists
	1	X	1	All excluding wait lists

3.3 VeriFast

VeriFast is a deductive verifier for single-threaded and multithreaded C and Java [14] and has been applied to a number of interesting industry case studies including a Linux device driver and an embedded Linux network management component [17]. Proofs are performed modularly on a per-function basis, each with respect to a specification: a precondition and postcondition. Informally, a proof of a function establishes that *all* executions of the function starting in a state satisfying the precondition either terminate in a state satisfying the postcondition or never terminate.[9] In VeriFast the language used to express specifications is based on *separation logic* [16]: a logic for reasoning about resources and their ownership (or permission). Resources include both allocated objects in memory (such as the queue buffer storage, including its individual elements) and abstract notions like the ability to mask interrupts. Critically, for a thread to access a resource it must have the correct permission (e.g., read-only access or exclusive access) to do so. This is a key check for ensuring memory safety and thread safety enforced automatically by VeriFast. Finally, VeriFast supports user-defined definitions which allow us to express the expected behavior of the queue for functional correctness. We illustrate these features by analyzing a proof in the next section.

4 ANATOMY OF A PROOF

Figure 4 gives a proof for a simplified version of the internal queue function `prvCopyDataToQueue`. This function is responsible for copying data into the queue and must only be invoked when the calling task or ISR has taken ownership of the queue by masking interrupts (note the comment on line 10) and the queue has space for the message. The implementation has three steps: (1) the message is copied to the back of the queue (the element pointed-to by `pcWriteTo`) (line 20), (2) this pointer is incremented by one element (line 25) modulo wraparound (lines 27–32) and (3) the number of messages is incremented (line 42).

The proof is the code of the function with a specification and proof annotations. The precondition (lines 4–6) and postcondition (lines 7–8) as well as the proof annotations are written inside special comments (`/*@ ... @*/`) ignored by compilation but visible to VeriFast. A high-level reading of this specification is that the function takes an arbitrary well-formed queue and returns the same queue with one new element at the back, whose contents is a copy of `pvItemToQueue`.

We begin by describing the definition of a well-formed queue (Figure 3) used in both the pre and postcondition. The definition uses N, M, W, R and K as we informally used them when describing the sequential operation of the queue (§3.2):

- N is the length of the queue (i.e., the value of the queue struct field `uxLength`)
- M is the size in bytes of each element (i.e., the value of the queue struct field `uxItemSize`)
- W is the *index* of the element pointed-to by the back of the queue pointer `pcWriteTo`

[9] This notion of correctness which ignores non-terminating executions is known as *partial correctness*. Checking for termination as well is known as *total correctness*. VeriFast supports both notions, but we do not verify termination in the queue proofs.

```
predicate queue(Queue_t *q, int8_t *Storage,
  size_t N, size_t M, size_t W, size_t R, size_t K,
  list<list<char> > abs) =
    // layout
    q->pcHead |-> Storage &*&
    q->pcTail |-> Storage + (N*M) &*&
    q->uxLength |-> N &*&
    q->uxItemSize |-> M &*&
    q->pcWriteTo |-> Storage + (W*M) &*&
    q->pcReadFrom |-> Storage + (R*M) &*&
    q->uxMessagesWaiting |-> K &*&
    // invariants
    0 < N &*& 0 < M &*&
    0 <= W &*& W < N &*&
    0 <= R &*& R < N &*&
    0 <= K &*& K <= N &*&
    W == (R + 1 + K) % N &*&
    buffer(Storage, N, M, ?contents) &*&
    length(contents) == N &*&
    // abstract representation
    abs == take(K, rotate_left((R+1)%N, contents))
```

Figure 3: Simplified definition of queue well-formedness. The notation |-> is read as "points-to".

- R is the *index* of the element pointed-to by the front of the queue pointer `pcReadFrom`
- K is the number of valid elements in the queue (i.e., the value of the queue struct field `uxMessagesWaiting`)

For example, in Figure 2 state S7 has $N = 4$, $M = 1$, $W = 1$, $R = 1$ and $K = 3$. The question mark syntax (e.g., `?N`) binds the name to the value of the struct field.[10] Finally, `abs` is an abstract representation of the queue, which is how we capture functional correctness of the queue. It is the list of *valid* elements (i.e., a list of list of chars) in the queue allowing for wraparound obtained by rotating until the front of the queue is at the head and taking the first K elements of the contents of the queue buffer storage. For example, in Figure 2 state S7 has `contents [[E];[-];[C];[D]]` (where - stands for any value) and `abs` is obtained by rotating this list by $(R+1)$ mod $N = 2$ times and taking the first $K = 3$ elements: `[[C];[D];[E]]`. The ability to 'cast' between the concrete representation of an object (such as the queue buffer storage) and a mathematical definition (such as the `abs` list) is a valuable feature of deductive verification.

Suitably equipped with this definition, the precondition can be explained as follows. The first requirement is that the queue is well-formed and the calling task or ISR has exclusive ownership of the resource. Both conditions are specified by `queue(...)` on line 4. The second requirement is that the queue must have space in its buffer for the message: $K < N$ (line 4). Thirdly, the input parameter `pvItemToQueue` must be a pointer to a buffer of at least M bytes (to avoid out-of-bounds memory accesses). We express this with another definition `chars(pvItemToQueue, M, ?x)` on line 5 where `x` is the abstract list of M bytes. More subtly, to avoid undefined behavior when calling `memcpy`, it is essential that the

[10] VeriFast calls this pattern matching, a restricted form of existential quantification

destination (`pcWriteTo` pointing into the queue storage) and source (`pvItemToQueue`) are disjoint (i.e., non-aliased). This is expressed in the precondition using the *separating conjunction* `&*&` which specifies such disjointness by definition.[11] Finally, only for the purposes of simplifying the proof in this example, we require that the message to be placed to the back of the queue.

The postcondition reflects the obligations that the function must satisfy. Specifically, that the queue remains well-formed with one more valid element ($K + 1$) and an updated back pointer that takes into account wraparound behavior (i.e., the new element pointed-to by `pcWriteTo` is $(W + 1) \bmod N$) and the new abstract representation is the original abstract representation `abs` with a new element containing `x` appended, where `x` is the contents of the input parameter `pvItemToQueue`.

Next, we turn to the proof annotations. VeriFast establishes the proof by *symbolic execution*: the function body is executed symbolically starting from the symbolic state described by the precondition. At each statement, VeriFast ensures that the permissions necessary to execute the statement are present in the symbolic state. And at the function return, VeriFast ensures that the ending symbolic satisfies the postcondition. The proof annotations can be seen as rewritings of the symbolic state necessary to help VeriFast with its reasoning.

- The annotation on line 15 calling `split_element` is a lemma to rewrite the queue storage from being a buffer of N elements into three pieces: the element that we wish to update (i.e., W) in the `memcpy` and the 'prefix' and 'suffix' elements on either side. For example, in Figure 2 this lemma operating on state `S7` would yield a prefix of `[[E]]` and suffix of `[[C];[D]]`. This rewrite is necessary so that the `memcpy` has exactly the right permission for its destination. Subsequently, we rewrite these three pieces back into a single buffer using two further lemmas (line 22–23). All of the lemmas used in the queue proofs are themselves proved in VeriFast.
- The annotations between lines 24-39 reestablish the queue invariant for well-formedness between `pcWriteTo` and W after the pointer is incremented and, if necessary, wrapped-around (lines 27–31). In particular, we need basic lemmas about modulo arithmetic to establish the updated pointer now points-to element $((W + 1) \bmod N) * M$. Notice that we have to establish this fact in both the `if` and `else` branch cases in order to deduce that this always holds at function return.
- The annotations between lines 45–47 are also necessary to reestablish queue invariants for well-formedness. The most complicated of these is `enq_lemma`, which establishes that in-place updating the queue buffer using `memcpy` and incrementing the back pointer is equivalent to simply appending the new element to `abs`.

[11] The separating conjunction is also denoted simply as $*$. The assertion $P * Q$ means that the symbolic heap can be split into two *disjoint* parts such that one satisfies P and the other Q. The separating conjunction between the queue and `chars` resources in the precondition means the symbolic heap must be splittable into two disjoint parts such that the queue resource appears in one and the chars resource in the other.

Concurrent Behavior. Finally, we sketch how the proofs reason about the concurrency mechanisms outlined in Section 3.2. Essentially we encode Table 2 as *ghost state*: logical state visible to VeriFast but not part of the implementation. The two generic mechanisms, interrupt masking and scheduler suspension, are modeled as two ghost mutexes. Access to the interrupt masking mutex is given to both tasks and ISRs, however, their lock invariant (the resources gained by the caller by acquiring the mutex) differ: A task that masks interrupts (modeled by acquiring the interrupt masking mutex) gains access to the queue including the ability to lock through `cTxLock` and `cRxLock`. An ISR that masks interrupts gains access to the queue and, if `cTxLock` and `cRxLock` are *unlocked*, the task waiting lists. Access to the scheduler suspension mutex is given only to tasks. A task that suspends the scheduler (modeled by acquiring the scheduler suspension mutex) gains access to the task waiting lists if `cTxLock` and `cRxLock` are *locked*. For each concurrent queue API function we specify the permissions necessary to use these two mechanisms and rely on VeriFast's permission checking to ensure that resources are correctly acquired and released.

5 FINDINGS

In this section we outline four findings discovered during verification. We note that FreeRTOS is heavily tested and the kernel has included the queue implementation from V1.0.0, so it is a testament to the thoroughness of formal verification that we were able to discover and report these findings in such battle-hardened code. For each finding we are able to prove that appropriate fixes resolve the underlying issue.

Unsigned wraparound in `xQueueGenericCreate`. This function is responsible for allocating a fresh queue in memory. The size of the queue buffer is `uxQueueLength * uxItemSize` where each is of type `UBaseType_t` (an unsigned integer of the natural width of the target platform). With large values this multiplication can cause unsigned integer wraparound resulting in a smaller buffer being allocated than expected. If the queue is subsequently used then the mismatch between the queue length and the underlying buffer would be a memory safety issue. This behavior is unlikely in practice because it involves requesting more memory than a device could address. This finding was addressed by adding an assertion check in https://github.com/FreeRTOS/FreeRTOS-Kernel/pull/75.

Signed overflow of `cTxLock` *and* `cRxLock`. As discussed in Section 3.2, the queue lock variables are used to protect the queue task wait lists. If the following behavior occurs then it is possible to cause signed overflow of these variables, which is undefined C behavior [13, 6.5 para 5].

- Initially, the queue is empty with both `cTxLock` and `cRxLock` unlocked (i.e., set to -1)
- A task calls `xQueueReceive` The queue is empty so the task will be blocked. The task atomically sets the queue lock variables to 0 (locked).
- An interrupt occurs and its corresponding ISR calls `xQueueGenericSendFromISR`. This function finds the queue locked so cannot access the task wait lists, but can increment the `cTxLock` value.

```
void prvCopyDataToQueue( Queue_t * const pxQueue,
                         const void * pvItemToQueue,
                         const BaseType_t xPosition )
/*@requires queue(pxQueue, ?Storage, ?N, ?M, ?W, ?R, ?K, ?abs) &*& K < N &*&
    chars(pvItemToQueue, M, ?x) &*&
    xPosition == queueSEND_TO_BACK;@*/
/*@ensures queue(pxQueue, Storage, N, M, (W+1)%N, R, K+1, append(abs, {x})) &*&
    chars(pvItemToQueue, M, x);@*/
{
    /* This function is called from a critical section. */
    UBaseType_t uxMessagesWaiting = pxQueue->uxMessagesWaiting;

    /* The abstract list of list of chars of `Storage` is `contents` */
    /*@assert buffer(Storage, N, M, ?contents);@*/
    /*@split_element(Storage, N, M, W);@*/
    /*@assert
        buffer(Storage, W, M, ?prefix) &*&
        chars(Storage + W * M, M, _) &*&
        buffer(Storage + (W+1) * M, (N-1-W), M, ?suffix);@*/
    memcpy( ( void * ) pxQueue->pcWriteTo, pvItemToQueue, ( size_t ) pxQueue->uxItemSize );
    /* After the update we stitch the buffer back together */
    /*@join_element(Storage, N, M, W);@*/
    /*@combine_list_update(prefix, x, suffix, W, contents);@*/
    /*@mul_mono_l(W, N-1, M);@*/
    pxQueue->pcWriteTo += pxQueue->uxItemSize;

    if( pxQueue->pcWriteTo >= pxQueue->u.xQueue.pcTail )
    {
        /*@div_leq(N, W+1, M);@*/ /* now we know W == N-1 so (W+1)%N == 0 */
        pxQueue->pcWriteTo = pxQueue->pcHead;
    }
    else
    {
        /*@{
        div_lt(W+1, N, M); // now we know W+1 < N
        mod_lt(W+1, N);    // so, W+1 == (W+1)%N
        note(pxQueue->pcWriteTo == Storage + ((W+1) * M));
        note(                      Storage + ((W+1) * M) == Storage + (((W+1) % N) * M));
        }@*/
        mtCOVERAGE_TEST_MARKER();
    }
    pxQueue->uxMessagesWaiting = uxMessagesWaiting + ( UBaseType_t ) 1;

    /*@{
    enq_lemma(K, (R+1)%N, contents, abs, x);
    mod_plus_one(W, R + 1 + K, N);
    mod_plus_distr(R+1, K, N);
    }@*/
}
```

Figure 4: A proof of a simplified version of `prvCopyDataToQueue`, which places the contents of the buffer `pvItemToQueue` into the queue. We have simplified the implementation and proof in this example by specializing the placement of the message to the back of the queue. The full proof covers all options (copy-to-front and overwrite) supported by `xPosition`.

- Assume the interrupt remains high so the task is never returned to and the ISR continues to call `xQueueGenericSendFromISR`. Then after enough further interrupts, the variable `cTxLock` will reach 127 and subsequently overflow.

This behavior is unlikely in practice because of the precise timing and conditions required. This finding was addressed by adding an assertion check in https://github.com/FreeRTOS/FreeRTOS-Kernel/pull/75.

Constructing an out-of-bounds pointer in `prvCopyDataToQueue`. As discussed in Section 4, this function inserts data into the queue. The parameter `xPosition` determines whether the insertion is to the back or front. In the latter case, after copying in the message, the pointer `pcReadFrom` is updated as follows:

```
pcReadFrom -= pxQueue->uxItemSize;
if( pcReadFrom < pxQueue->pcHead )
{
    pcReadFrom = pxQueue->pcTail - pxQueue->uxItemSize;
}
```

In the case where `pcReadFrom` initially points-to the zeroth element of the buffer (i.e., equal to `pcHead`) then the decrement by `uxItemSize` (which must be non-negative) results in a pointer that is out of the bounds of the queue object. This is undefined behavior (even though the resulting pointer is never dereferenced and even though the pointer is only temporarily out-of-bounds) [13, 6.5.6 para 8]. As an example of how this undefined behavior might manifest: a compiler could assume that the pointer arithmetic always results in a pointer into the queue buffer (otherwise the result is undefined). As such, the if statement is redundant and the compiler could remove it as dead code. No compiler that we are aware of, including those used by FreeRTOS kernel ports[12], currently takes such an aggressive optimization.

Non-terminating execution in `xQueueReceive`. In the queue API function `xQueueReceive` there is a case corresponding to the behavior when: (i) The queue is empty, (ii)`xTicksToWait`, the amount of time the calling task is willing to block, is non-negative and (iii) a timeout has occurred (signaled by the task utility function `xTaskCheckForTimeOut` returning `pdTRUE`). This case is line 1399 in `queue.c` (V10.4.3). Subsequently the function checks *again* whether the queue remains empty and if so returns an error (line 1450) or falls-through to retry receiving from the queue (line 1454). This fall-through behavior permits the following non-terminating behavior involving a task and two interrupts:

- Initially, the queue is empty
- A task calls `xQueueReceive` with 0 < `xTicksToWait`
- **Loop**
  - The condition above occurs since the queue is empty, `xTicksToWait` is non-negative and the function `xTaskCheckForTimeOut` returns `pdTRUE`. Crucially, prior to us reporting this finding, `xTaskCheckForTimeOut` had a path that returned true *without* decrementing `xTicksToWait`. That is, `xTicksToWait` remains non-negative.
  - An interrupt occurs and its corresponding ISR calls `xQueueGenericSendFromISR` so that the queue is no longer empty.
  - Back in the task, the function checks *again* whether the queue remains empty. It is not, so the function falls-through to retry receiving from the queue.
  - An interrupt occurs and its corresponding ISR calls `xQueueReceiveFromISR` so that the queue returns to empty. This returns us to the loop state.

This behavior would be difficult to trigger in practice due to the precise timing required between the task and interrupts. This finding was addressed by ensuring that `xTaskCheckForTimeout` sets `xTicksToWait` to 0 whenever it returns true in https://github.com/FreeRTOS/FreeRTOS-Kernel/pull/82.

[12] https://freertos.org/RTOS_ports.html

6 CONCLUSIONS

We have reported on the formal verification of memory safety, thread safety and functional correctness of the FreeRTOS concurrent queue implementation using the VeriFast deductive verifier. In doing so, we have raised the level of assurance for the FreeRTOS interprocess communication mechanism which uses this data structure. Our proofs are publicly available and run as part of continuous integration to help ensure their continued maintenance. This work is part of a larger trend in FreeRTOS to use formal verification techniques where they are most effective and the benefits merit the costs.

In terms of future work, we see two interesting directions. Firstly, linearizability is a standard correctness property for concurrent data structures [12]. We have a hand-proof of this property for the FreeRTOS queue using a forward-simulation argument; we would like to mechanize this result and link it to our VeriFast proofs. Secondly, now that we have proofs for the queue, we would like to evaluate the proof maintenance costs. On the positive side, these proofs give FreeRTOS developers a safety net as they seek to make changes. However, large changes could potentially require corresponding effort to repair the proofs. Optimizations to the queue implementation such as zero-copying would be an ideal case study.

ACKNOWLEDGEMENTS

We are grateful to Richard Barry for extensive discussions about FreeRTOS; and Daniel Schwartz-Narbonne, Mark R. Tuttle and Mike Whalen for valuable feedback throughout this work.

REFERENCES

[1] Amazon Web Services. *The FreeRTOS Reference Manual.* 2017.
[2] R. Barry. *Mastering the FreeRTOS Real Time Kernel: A Hands-On Tutorial Guide.* 2016.
[3] C. Blundell, E. C. Lewis, and M. Martin. Subtleties of Transactional Memory Atomicity Semantics. *IEEE Computer Architecture Letters*, 5(2), 2006. doi:10.1109/L-CA.2006.18.
[4] S. Cheng, J. Woodcock, and D. D'Souza. Using Formal Reasoning on a Model of Tasks for FreeRTOS. *Formal Aspects Computing*, 2015. doi:10.1007/s00165-014-0308-9.
[5] N. Chong, B. Cook, K. Kallas, K. Khazem, F. R. Monteiro, D. Schwartz-Narbonne, S. Tasiran, M. Tautschnig, and M. R. Tuttle. Code-Level Model Checking in the Software Development Workflow. In *International Conference on Software Engineering, Software Engineering in Practice*, 2020. doi:10.1145/3377813.3381347.
[6] E. M. Clarke, O. Grumberg, D. Kroening, D. Peled, and H. Veith. *Model Checking, Second Edition.* MIT Press, 2018.
[7] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A Practical System for Verifying Concurrent C. In *International Conference on Theorem Proving in Higher Order Logics TPHOLS*, 2009. doi:10.1007/978-3-642-03359-9_2.

[8] B. Cook, K. Khazem, D. Kroening, S. Tasiran, M. Tautschnig, and M. R. Tuttle. Model Checking Boot Code from AWS Data Centers. In *Computer Aided Verification CAV*, 2018. doi:10.1007/978-3-319-96142-2_28.

[9] D. Déharbe, S. Galvão, and A. M. Moreira. Formalizing FreeRTOS: First Steps. In *Brazilian Symposium on Formal Methods SBMF*, 2009. doi:10.1007/978-3-642-10452-7_8.

[10] S. Divakaran, D. D'Souza, A. Kushwah, P. Sampath, N. Sridhar, and J. Woodcock. Refinement-Based Verification of the FreeRTOS Scheduler in VCC. In *International Conference on Formal Engineering Methods ICFEM*, 2015. doi:10.1007/978-3-319-25423-4_11.

[11] J. F. Ferreira, C. Gherghina, G. He, S. Qin, and W. Chin. Automated verification of the FreeRTOS scheduler in Hip/Sleek. *International Journal on Software Tools for Technology Transfer*, 2014. doi:10.1007/s10009-014-0307-4.

[12] M. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12, 1990. doi:10.1145/78969.78972.

[13] ISO/IEC. *Programming languages – C*. International standard 9899:2011, 2011.

[14] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods NFM*, 2011. doi:10.1007/978-3-642-20398-5_4.

[15] J. T. Mühlberg and F. Leo. Verifying FreeRTOS: from requirements to binary code, 2011.

[16] P. W. O'Hearn. Separation logic. *Communications of the ACM*, 2019. doi:10.1145/3211968.

[17] P. Philippaerts, J. T. Mühlberg, W. Penninckx, J. Smans, B. Jacobs, and F. Piessens. Software Verification with VeriFast: Industrial Case Studies. *Science of Computer Programming*, 82, 2014. doi:10.1016/j.scico.2013.01.006.

[18] C. K. Pronk. Verifying FreeRTOS; a feasibility study. Technical report, Delft University of Technology, 2010.

[19] T. Ringer, K. Palmskog, I. Sergey, M. Gligoric, and Z. Tatlock. QED at Large: A Survey of Engineering of Formally Verified Software. *Foundations and Trends in Programming Languages*, 2019. doi:10.1561/2500000045.

[20] D. Sanán, L. Yang, Y. Zhao, Z. Xing, and M. Hinchey. Verifying FreeRTOS' Cyclic Doubly Linked List Implementation: From Abstract Specification to Machine Code. In *International Conference on Engineering of Complex Computer Systems ICECCS*, 2015. doi:10.1109/ICECCS.2015.23.

[21] F. Vogels, B. Jacobs, and F. Piessens. Featherweight VeriFast. *Logical Methods in Computer Science*, 2015. doi:10.2168/LMCS-11(3:19)2015.