FORMALLY MODELLING AND VERIFYING THE FREERTOS REAL-TIME OPERATING SYSTEM

Shu Cheng

Doctor of Philosophy

University of York
Computer Science

September, 2014

Abstract

Formal methods is an alternative way to develop software, which applies mathematical techniques to software design and verification. It ensures logical consistency between the requirements and the behaviour of the software, because each step in the development process, i.e., abstracting the requirements, design, refinement and implementation, is verified by mathematical techniques. However, in ordinary software development, the correctness of the software is tested at the end of the development process, which means it is limited and incomplete. Thus formal methods provides higher quality software than ordinary software development. At the same time, real-time operating systems are playing increasingly more important roles in embedded applications. Formal verification of this kind of software is therefore of strong interest.

FreeRTOS has a wide community of users: it is regarded by many as the *de facto* standard for micro-controllers in embedded applications. This project formally specifies the behaviour of FreeRTOS in Z, and its consistency is verified using the Z/Eves theorem prover. This includes a precise statement of the preconditions for all API commands. Based on this model, (a) code-level annotations for verifying task related API are produced with Microsoft's Verifying C Complier (VCC); and (b) an abstract model for extension of FreeRTOS to multi-core architectures is specified with the Z notation.

This work forms the basis of future work that is refinement of the models to code to produce a verified implementation for both single and multi-core platforms.

Contents

List of Figures

List of Tables

List of Schemas

Acknowledgements

First and foremost, I would like to thank my supervisor, Prof. Jim Woodcock, whose patient and continuous guidance, constructive suggestions and kind encouragement helped me throughout my PhD. It has been great pleasure to work with him.

I also want to express my great gratitude to my parents and family. Without their boundless love, teaching and guidance I would never be who I am.

And I also want to thank my colleagues in the Computer Science Department at the University of York, who have contributed to discussions and support about my work. Especially, I appreciate Steve King, Rob Alexander, Jeremy Jacob, Sam Simpson, Victor Bandur, Pakorn Waewsawangwong, Tasawer Khan, Chris Poskitt, James Williams, and Detlef Plump. Thanks are also due to members of the Indian Institute of Science in Bangalore for their helpful discussions and comments of this work: thanks to Deepak D'Souza, Sumesh Divakaran, Anuridh Kushwah, Virendra Singh, and Nigamanth Shridar.

I also want to thank my assessors, Steve King and Jonathan Bowen. Their precious feedback helps me to make the thesis much more better.

Last, but not least, I want to appreciate all my friends: especially to Lin Liu, Siyu Wang, Lei Chen, Hao Sun, YunFeng Ma, Hengyu Wu, Wen Luo and Keshu Xu. Their friendship, help and encouragement, always support me, even in the hardest times, and bring me the most blissful memories.

Declaration

The work in this thesis has been carried out by the author between September 2010 and September 2014 at the Department of Computer Science, University of York. This work has not previously been presented for an award at this, or any other, University. Apart from work whose authors are clearly acknowledged and referenced, all other works presented in this thesis were carried out by the author. First section of Chapter 4 of this thesis have appeared in previously published paper:

- S. Cheng, J. Woodcock, and D. D'Souza, Using formal reasoning on a model of tasks for FreeRTOS. *Formal Aspects of Computing*, pp. 1–26, 2014. [Online]. Available: DOI:10.1007/s00165-014-0308-9

  All the technical works presented in this paper are carried out by the author of this thesis.

Chapter 1

INTRODUCTION

This chapter begins by introducing formal methods and the international Verified Software Initiative (VSI). It then introduces FreeRTOS, as formally verifying the correctness of FreeRTOS is one of the pilot projects of the VSI. Next, this chapter introduces VCC and shows that it is possible to combine formal specification and code level annotations together to verify the source code. It then discusses the limitations of single core processors and the benefits of multi-core processors. In addition the objectives and challenges of verifying FreeRTOS are clarified. Finally, the structure of the thesis is given.

1.1 Formal Methods

Formal methods apply mathematical techniques to software design and verification and are normally supported by tools [1]. The general development process for formal methods is:

(a) Use mathematical expressions to specify the state and the behaviour of the software according to the documented requirements, which can generally be expressed by state transitions. An abstract specification will be produced in this step;
(b) Apply mathematical theorems and lemmas to verify the specification or model;

(c) Refine the specification from abstract level to concrete model;
(d) Repeat steps b & c until executable code is generated. Note, the relation between each refinement also has to be verified.

Formal methods can provide higher quality software than ordinary software development, because all models produced in the development process can be verified and proved using mathematical logic. This can efficiently detect any faults in the software at the initial stage of development, which may later have lead to huge losses. For instance, an error in the Inertial Reference System (IRS) of Ariane 5 caused the explosion of the rocket in June 1996 [2], costing around half a billion US dollars. The US Department of Commerce also estimates that the losses caused by avoidable software errors is between 20 and 60 billion dollars every year [3, 4]. Without formal methods these kinds of faults are sometimes very hard to discover, and even if revealed, may be too expensive to correct at a later stage [5].

However, formal methods have not been widely applied in industry, although they have significant advantages. Hall [6] believes that there are seven common myths or misunderstandings about formal methods, some of which cause this situation. Firstly, it is thought that formal methods increase the cost of the development. Yet Hall [6] indicates that in his experience, applying formal methods in commercial projects decreases the development cost. Although there is a one-time cost for learning the non-user-friendly tools which often support formal methods, the developer gains more benefits from the reduction of cost in the amount of testing and maintenance. King *et al.* [7] claim that using formal methods is more efficient for detecting faults than the most efficient testing phase, which also increases the cost. The cost of verifying and testing software may occupy 30% to 50% of the total cost of a software project [8]. This can increase to 70% for hardware. Even with this huge investment, however, Dijkstra [9] believes testing can never guarantee that software is free of bugs. Secondly, “formal methods involve complex mathematics and are incomprehensible to clients” [6]. Although formal methods apply mathematical technology in documentation and design, it only needs knowledge related to logic and set theory, which is a fundamental part of mathematics. Meanwhile,

using formal methods, developers may experiment with the model and demonstrate it to clients using animation. Such animations can clearly show clients the behaviour of the system. Formal methods help developers to organise documentation much better and because of the mathematical rigour, the documentation is also more likely to be unambiguous and precise. This makes it easier for clients to use and understand the system [4]. Lastly, people believe that formal methods are only used in academic and research fields or in highly critical systems. However, it has been reported that formal methods are suitable for industrial-scale applications [6, 7, 10, 11]. Moreover, Berry [12], chief scientist at ESTEREL Technologies, shows that the control system of the Airbus A380, which has five million lines of code, was automatically generated by formal methods and all worked first time.

Fortunately, this situation is changing. In 2003, Hoare [13] suggested the international Grand Challenge for Computing Research to build a verifying compiler, which could automatically verify whether a program met its requirements [14]. Based on this idea, the international Verified Software Initiative (VSI) [4, 5, 13, 15], led by Hoare, was proposed. The main aim of the VSI is to work out a more approachable strategy for verifying software with the integrated support tools. Several pilot projects have been selected for VSI, such as the Mondex electronic purse [16], POSIX file store [17], etc.

1.2 FreeRTOS

As a widely used real-time operating system, the function of FreeRTOS can be divided into three large categories: (a) multitasking task management; (b) inter-task communication and synchronisation; and (c) memory management, interrupt management and other features. The three key elements of FreeRTOS are:

Tasks: user processes.

Queues: communication mechanisms between tasks and interrupts.

Semaphores and Mutexes: the facilities which are used for resource management, event counting, mutual exclusion locks, etc.

1.2.1 Task Management

Tasks in FreeRTOS can be regarded as occupying one of two top-level states, `running` or `notRunning`. The running task is recorded by the task control block handler `pxCurrentTCB` and simply indicates that the task is currently executing on the processor. The `notRunning` state can be decomposed into three sub-states: `ready`, `suspended`, and `blocked`. The following lists in FreeRTOS are used to manage this:

Ready Lists (`pxReadyTasksLists`) this is an array of the task lists, in which tasks are available to be scheduled to the `running` state.

Delay List (`xDelayedTaskList1`) & Overflow Delay List (`xDelayedTaskList2`) tasks in these lists are blocked by an event for a certain period. They are sorted by wake-up time. Because the time is expressed by ticks in FreeRTOS, if the wake-up time is later than the time represented by $maximum\_delay\_ticks - current\_ticks$, the ticks for wake-up time could overflow. Therefore, an overflow-delay list is required.

Suspended List (`xSuspendedTaskList`) tasks in this list have been suspended, and wait until they are resumed by another task.

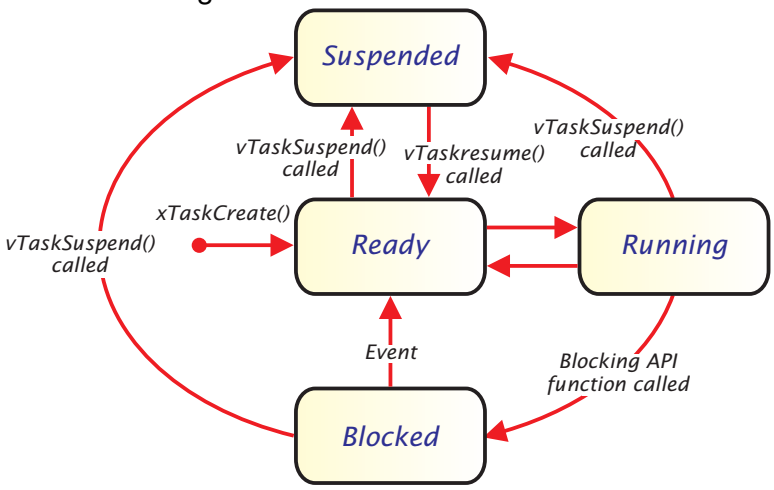
Pending Ready List (`xPendingReadyList`) tasks resumed from Interrupt Service Routines (ISRs) are kept in this list temporarily, while the scheduler is not running.

Waiting Termination List (`xTaskWaitingTermination`) deleted tasks stay here and wait to be removed by the `idle` task.

Tasks transit between these states as described in Fig. 1.1. For instance, a task cannot directly transit from `suspended` to `running`, because only `ready` tasks can be scheduled as `running` [18]

All tasks have their own priority, `uxPriority`, which is used by the scheduler. Tasks can have another priority, `uxBasePriority`, which records the original priority of

Figure 1.1: State chart for Tasks



tasks when priority inheritance occurs. The scheduler is responsible for counting the clock ticks, used to express time, and schedules the tasks. The scheduling policy adopted here is priority-based scheduling, which means that the task with the highest priority and in the ready state can be executed. As a result, it is impossible to use FreeRTOS in hard real-time environments. When a ready task has a higher priority than the running task, it will displace the running task from the CPU. The scheduler has two ways of switching tasks: pre-emptive and cooperative scheduling. In pre-emptive mode, the task with the highest priority will block the running task immediately and take the CPU. In cooperative mode, the running task can finish its CPU time before the task with the highest priority takes over. API functions are provided for task creation, deletion, and control. It is worth noting that the deletion API function does not actually delete a task from the system: it only adds the task to `xTasksWaitingTermination` and removes its reference from related task lists. The `idle` task, with permanent priority 0, the lowest priority, is used to do the deleting job and release the memory allocated by the kernel. However, it does not collect the memory allocated by the user, so tasks have to release used memory themselves, before being deleted.

In our specification, the function $state$, which is a total function from $TASK$ to $STATE$, in schema $StateData$ (see Sect. 4.2.1, Page. 43) is used to specify the states of tasks. Further, the reverse function of $state$ can be used to cal-

Figure 1.2: An example application that uses RTOS (Task related).

```
1  xTaskHandle txh1;
2
3  void tx1(void * xPara){
4  xTaskCreate(tx3, (signed char *) "Task_3", 1000, NULL, 4, NULL);
5  for( ;; );
6  }
7
8  void tx2(void * xPara){
9  for( ;; ){
10 vTaskPrioritySet(txh1, 3);
11 }
12 }
13
14 void tx3(void * xPara){
15 for( ;; ){
16 vTaskDelete(NULL);
17 }
18 }
19
20 int main(void){
21 xTaskCreate(tx1, (signed char *) "Task_1", 1000, NULL, 1, & txh1);
22 xTaskCreate(tx2, (signed char *) "Task_2", 1000, NULL, 2, NULL);
23
24 vTaskStartScheduler();
25 return 0;
26 }
```

culate tasks in a specific state; for instance, `pxReadyTasksLists` can be represented by $state^{\sim}(\!|\ \{ready\}\ |\!)$. This also works for running tasks: the result of $state^{\sim}(\!|\ \{running\}\ |\!)$ is a set with only one element—$running\_task$, which represents the handler `pxCurrentTCB`. The function $priority$ in schema $PrioData$ and $old\_priority$ in schema $OriginalPrioData$ of Mutex model represents tasks' `uxPriority` and `uxBasePriority` respectively. They are defined in Chap. 4.

We use a simple example application to illustrate the functionality provided by FreeRTOS. Fig. 1.2 shows the C code of an application that uses the FreeRTOS API function related to task management. Initially, the application creates two tasks: `Task1` and `Task2`, with priority 1 and 2 respectively (a higher number indicates higher priority), and then starts the FreeRTOS scheduler. The scheduler then runs `Task2`, which immediately increases the priority of `Task1` to 3. `Task2` is now pre-empted by `Task1`, which gets to execute and creates a new task—`Task3` with priority 4, which is the highest at the moment. Therefore, it pre-empts `Task1` and can execute. Once `Task3` is executing, it deletes itself, which triggers the scheduler to reschedule the system. As `Task1` has the highest priority at this moment, it gets to execute again and will continue to execute.

We now describe in more detail what happens in the FreeRTOS implementation code. The application code for `main`, `tx1`, `tx2` and `tx3` is compiled along with the FreeRTOS code (for the scheduler and the API function calls including `xTaskCreate`) and loaded into memory. The scheduler code is loaded into the Interrupt Service Routine (ISR) code area so that it services software interrupts.

By analysing the source code of FreeRTOS, we see that execution begins with the first instruction in `main`, which is the call to the `xTaskCreate` API function. This code is provided by FreeRTOS. It allocates 1 kilobyte (defined in the parameters of `xTaskCreate`) of memory from the heap to the task stack, as well as space to store its Task Control Block (TCB) [19, 18]. From the source code, we see that the TCB contains all vital information about the task: where its code (`tx1` in this case) is located, where its stack begins, where its current top-of-stack pointer is, what its priority is, and so on. The API function call initialises the TCB entries for `Task1`. It then creates and initialises the various lists that the OS maintains, such as `pxReadyTasksLists`, `xSuspendedTaskList` and so on. It finally adds `Task1` to the ready list and returns. Next, `main` calls `xTaskCreate` for `Task2` and the API function call sets up the stack and TCB for `Task2` and adds it to the ready list, in a similar way. The next instruction in `main` is a call to the `vTaskStartSchedular` API function, which is also provided by FreeRTOS [19, 18]. This call creates the `idle` task with priority 0, and adds it to the ready list. It also sets the timer tick interrupt to occur at the required frequency. Finally, it does a context-switch to the highest priority ready task (i.e., it restores its execution state, namely the contents of its registers, from the task's stack where they were stored). The processor will next execute the instruction in the task that is resumed. In our example, this means that `Task2` will now begin execution.

When `Task2` begins execution it makes an API function call to `vTaskPrioritySet`. The code for this API function call compares the new priority and the current priority to decide whether scheduling is needed. If the API function increases the priority of a task or decreases the priority of the current running task, a reschedule will be requested. It then assigns the new priority to the target task, and moves the task to

the proper position in the ready list, if it is a ready task. In our case, the priority of Task1 is changed to 3, it is moved to the correct position in pxReadyTasksLists. The API function code then execute a yield (a kind of software interrupt) that is trapped by the scheduler. The scheduler picks the highest priority ready task, which in this case is Task1, and makes it the running task. Before this, the scheduler saves the registers of Task2 to its stack, and restores the register context of Task1 from its stack.

Task1 now creates the new task Task3. The process is similar to the xTaskCreate call to create Task1 and Task2. The difference is that here xTaskCreate triggers scheduling to make Task3 run.

When Task3 begins execution, it makes a call to the vTaskDelete API function. The code for this API function is simple. It removes the target task from the state list and related events list; in this case, Task3 is removed from pxReadyTasksLists. As it is the current running task, the API function code triggers scheduling again to make the highest priority ready task run, which is Task1. Task1 then executes its trivial for-loop, *ad infinitum*.

The animation and formal verification of our specification for this process will be illustrated in Chap. 7.

1.2.2 Communication and Synchronisation

In FreeRTOS, tasks and interrupts communicate and synchronise with each other through queues. When two tasks in FreeRTOS need to exchange information, they send and receive information to and from a queue. As items are exchanged between tasks and queues by being copied to or from a queue, the size of each item in the queue must be the same. Otherwise, when a task receives an item from a queue, it would be confused as to how many bytes needed to be received. Every queue fixes the size of all items it can receive using uxItemSize. All queues also have a capacity, uxLength, which indicates how many items can be held by the queue. The number of items currently stored in the queue needs to be

recorded as well. Tasks will be blocked while they attempt to send (receive) items to (from) a full (empty) queue. The following two sequences are used to manage this: (a) `xTasksWaitingToSend` records tasks blocked by sending operations; and (b) `xTasksWaitingToReceive` records tasks blocked by receiving operations. As well as these basic properties, a number of other fields are recorded also for a queue, such as, `pcHead` and `pcTail`, which represent where the queue starts and ends.

Semaphores and mutexes, which are used to manage resources, mutual exclusion locks and so on, are implemented by queues. They are considered to be special queues. Specifically, the item size for semaphores and mutexes is 0. This is because, when tasks take a semaphore or mutex, they do not copy items from semaphores and mutexes. What is of interest to the task which attempts to take the semaphore or mutex, is whether it is available or not. The initial state of semaphores and mutexes is full rather than of empty, which is the initial state for normal queues. The main difference between semaphores and mutexes is that the maximum length for mutexes is always 1; on the another hand, the size of semaphores can be any unsigned number. Moreover, mutexes support the priority inheritance mechanism when a higher priority task is waiting to take a mutex which is hold by a lower priority task. Each mutex has its own mutex holder if it is taken by a task. The holder of a mutex can repeatedly take it at any time. Therefore, a mutex needs to know who is its holder. It overrides the field `pcTail` of normal queue to `pxMutexHolder` for this purpose.

In our specification, functions q_max and q_size in the $QueueData$ schema of the $Queue$ model (see Sect. 4.3, Page. 63) are used to represent the capacity and the current size of the queue respectively. Functions $wait\_snd$ and $wait\_rcv$ are used to indicate the blocked task for each queue in the system. Meanwhile, mutex related information is included in the $Mutex$ model (see Sect. 4.5, Page. 79). For instance, the function $mutex\_holder$, which represents the `pxMutexHolder`, is contained by $MutexData$ schema of $Mutex$ model. They are defined in Chap. 4.

Figure 1.3: An example application that uses RTOS (Communication related).

```
1  xTaskHandle tskH;
2  xSemaphoreHandle xMutex;
3
4  void tx1(void * xPara){
5  xSemaphoreTake(xMutex, portMAX_DELAY);
6  for( ;; );
7  }
8
9  void tx2(void * xPara){
10 vTaskDelay(10);
11
12 xSemaphoreTake(xMutex, portMAX_DELAY);
13 for( ;; );
14 }
15
16 int main(void){
17 xMutex = xSemaphoreCreateMutex();
18
19 xTaskCreate(tx1, (signed char *) "Task_1", 1000, NULL, 2, &tskH);
20 xTaskCreate(tx2, (signed char *) "Task_2", 1000, NULL, 3, NULL);
21
22 vTaskStartScheduler();
23 return 0;
24 }
```

Similar to Sect. 1.2.1, we use a simple example application (Fig. 1.3) to illustrate functionality related to communication in FreeRTOS. Initially, the application creates a mutex, `xMutex` and two tasks: `Task1` and `Task2`, with priority 2 and 3 respectively and then starts the scheduler, which runs `Task2`. It requests to delay for 10ms, which blocks `Task2` and lets `Task1` execute. Once `Task1` is executing, it takes the mutex `xMutex` then executes its infinite loop. After 10ms, `Task2` wakes up. As it has higher priority than `Task1`, it preempts `Task1` and starts to execute. `Task2` also tries to take the mutex, `xMutex`. However, it has been held by `Task1`. Therefore, `Task2` is blocked for `portMAX_DELAY` and `Task1` can execute again.

In detail, `main` creates `xMutex` by calling `xSemaphoreCreateMutex` (Note, the operations, such as load code, create task, etc., which have been described in Sect. 1.2.1, are not repeated here). As declared in *semphr.h*, `xSemaphoreCreateMutex` actually executes the code of `xQueueCreateMutex`, which is defined in *queue.c* [19, 18]. This code allocates space to store the new queue structure (`xQueue`) and initialise the structure for `xMutex`. For instance, set the type to `queueQUEUE_IS_MUTEX`, set the holder of the mutex to `NULL`, set item size to 0, etc. Next, `main` creates `Task1` and `Task2` and starts the scheduler.

When `Task2` begins execution, it makes a call to the `vTaskDelay` API function.

The code for this API function call will add Task2 to an event list, which is a priority queue associated with the delayed tasks, with a value that corresponds to the current tick count plus 10. Then, Task1 is scheduled as the running task.

Once Task1 starts to execute, it calls xSemaphoreTake to take the mutex, xMutex. xSemaphoreTake is also declared in *semphr.h* and executes code in *queue.c*, which is xQueueGenericReceive. It checks whether there is an item available in the queue (i.e., xMutex). If there is, the calling task receives the item. Otherwise, it is blocked for a period which is specified by a parameter of the API function. As there is no task holding xMutex at the moment, Task1 can successfully take the mutex. It then executes its infinite for-loop, until an interrupt for the next timer tick arrives from the hardware clock. This interrupt is again trapped by the scheduler and it increments its tick count. The scheduler then checks if any of the delayed tasks have a time-to-awake value that equals the current tick count. There is none and the scheduler hands back control to Task1. However, when the 10th timer interrupt takes place, the scheduler finds that Task2's time-to-awake equals the current tick count, and moves it to the ready queue. Since there is now a higher priority ready task, Task1 is swapped out and Task2 is restored and made to execute. It then attempts to take the xMutex. As Task1 holds the xMutex at the moment, Task2 is blocked by the mutex. The event list item of Task2 is added to the xTaskWaitingToReceive of xMutex as well. Therefore, Task1 can execute again and stay in an infinite for-loop.

This process will also be animated and formally verified in Chap. 7.

1.2.3 Other API functions

FreeRTOS also provides an API function for other operations, such as memory management, interrupt management, etc. Memory management related API function calls can be used to allocate and free memory. When a task or queue needs memory, pvPortMalloc can be used to do this. It first locates one of the available memory blocks, and then returns its pointer to the task or the queue. To release memory, vPortFree can be used. Meanwhile, interrupt related API functions can

be used to serve interrupts, enter/exit critical sections and so on. Specifically, all the interrupts have a piece of server code, an Interrupt Service Routine (ISR). When the operating system services the interrupt, it cannot accept another interrupt. Furthermore, as a real-time operating system, some parts of the code may be critical, which means they are unable to be interrupted. When the program enters this section of code, the counter `uxCriticalNesting` would be increased and `portDISABLE_INTERRUPTS` is called to set a processor flag to refuse further interrupts. When it exits the critical section, the counter is decreased. At this time, the value of the counter will be checked. If it is greater than zero, the processor flag remains the same, refusing further interrupts. Otherwise, if and only if it decreases to zero, `portENABLE_INTERRUPTS` can be applied to reset the flag to enable interrupts.

1.3 VCC

The Verifying C Complier (VCC) was developed by Microsoft for the Hypervisor Verification Project [20]. It verifies the correctness of annotated C programs. Annotations used for VCC include function specifications, state assertions, type invariants and so on [21]. As described above, normally, there are several steps of refinement and verification from the abstract model to the concrete specification and the executable code. Sometimes these are difficult and expensive to perform. Using the specifications from verified abstract models to directly verify the C code can be interesting and efficient.

Using Microsoft Visual Studio (MVS) for VCC is recommended. With macro definitions, the normal C compiler in Visual Studio can ignore the annotations used by VCC. On the other hand, the VCC verifier may use the C code and annotation together. They are translated into Boogie [22] files. These files are then used to generate *.sx files, which can be used by the Z3 prover [23]. VCC translates the C code and the annotation to mathematical formulas and verifies them using the Z3 prover, rather than analysing the code and looking for bugs. Once a piece of code has been verified by VCC, its correctness with respect to the preconditions

and post conditions can be guaranteed. Verification in VCC is modular. It does not go through every function call to verify the code of a function. Instead, it verifies a function with the information of a function contract for each called function. VCC assumes that the called functions are correct. In this case, it only needs to verify that when the function call happens the system state satisfies the preconditions of the called function. If it does, VCC knows that the post condition of the called function is satisfied as well. With this feature, developers can verify a function, even when its sub-functions are not finished or verified. In addition, with the benefits of the Z3 prover, VCC provides the Model Viewer which shows an example for each failure, when Z3 fails to verify the code. These examples contain a sequence of the system states which lead to the failure. This is helpful for the developer to understand why the code failed to verify.

1.4 Multi-core Processor

The Central Processing Unit (CPU) is the core component of the computer. Its performance determines the performance of the whole computer system. Therefore, the hardware industry has continued to try to improve the performance of the processor. There are two common ways to achieve this:

(a) Increasing the number of transistors on the chip. The Intel 4004, the first microprocessor built in 1971, had 2,300 MOS transistors [24]. According to Moore's Law, the number of transistors on a single chip will double approximately every two years [25]. Thus, after around 50 years development, it is now possible to put more than 500 million transistors on a single chip, e.g. the Intel i7-680UM Processor [26]. Due to the large number of transistors, an increasing number of resources are now available on a chip and processors have become progressively more powerful.
(b) Increasing the clock rate of the processor.

To use and control the resources on a single core processor efficiently, a large number of complex circuits have been designed and used. Because of this, design

and verification for a processor based on traditional single core architecture is increasingly difficult. Bose *et al.* have reported that verification activities can take up around 70% of the net development cost [27]. Meanwhile, increasing the clock rate is one of the direct ways to improve the performance. Nevertheless, this is limited by the physical features of the processors, power consumption and related thermal problems, which have also become ever more critical [27]. After every pulse of the clock, each transistor needs to take some time to transfer to a new state. If a clock pulse occurs before that, the data and the state of the processor will be incorrect, which is unacceptable. Furthermore, a higher clock rate means a higher power cost. Taylor *et al.* have indicated that it would increase power consumption by the increase in clock rate cubed [28]. In addition, the thermal issues also increase with energy expended. Due to these issues, speeding up the clock frequency to obtain higher performance has reached a bottleneck.

These issues are especially serious for embedded systems. This is because embedded systems are designed for a small number of dedicated functions [29] and they normally work in mobile systems and/or critical and real-time systems, such as sensor controllers and car control systems, etc. Due to the function of embedded systems, it is impossible to provide them with unlimited power.

Multi-core processors, which are composed of two or more independent cores on a chip, seem an alternative way to solve the problem. Multi-core processes are:

(a) Easy to design – due to parallel computation, multi-core processors can use several smaller and more simple cores to achieve a higher performance than a huge, complex core. Therefore, the designer only needs to repeat the simple cores across the chip and focus on the design of the communication method between separate cores, such as on-chip networks, bus and so on. Intel Research [30] reports that to design a single core chip with 100 million transistors would take about twice as long with twice as many people than a multi-core processor with the same number of transistors.

(b) Energy efficient – because multi-core processors can separate the task into

independent subtasks and share them with different cores on the chip, they gain high performance with a lower clock rate. As described above, the raising of the clock rate would lead to higher power-consumption and related thermal problems. With the reduction of the clock rate, these issues would be handled. For instance, the power of the Intel® Pentium® 4 Processor 531, which is a single core processor with a 3.00 GHz clock rate, is 84W [31]. However, the Intel Teraflops Research Chip, which is a multi-core processor with a 3.16 GHz clock frequency, consumes only 62W of power [30].

(c) Scalable – because the multi-core processor repeats the simple core across the chip, it is possible to place as many cores as the limits of the technology. In the laboratory, processors with 1,024 cores on a chip are now available.

Due to these benefits, an increasing number of companies use multi-core architecture.

1.5 Objectives and Challenges

1.5.1 Objectives

Our aim is to carry out a systematic exercise towards the verification of FreeRTOS that will:

(a) Produce a formal specification of its intended behaviour.
(b) Produce an annotated version of the implementation for VCC to verify.
(c) Identify aspects of its implementation that do not conform to this specification.
(d) Produce a detailed model of the core scheduling-related functionality that can serve as a basis for fixing the current implementation to obtain a “verified” version of FreeRTOS, engineered as originally intended by the developers.
(e) Produce an abstract model for a multi-core platform, which is an extension of FreeRTOS.

1.5.2 Structure View

Fig. 1.4 illustrates the whole structure of the project. To achieve these objectives defined above, three tools (Z/Eves, VCC and ProZ) are used in the project,

which are represented by rectangles in the figure. Specifically, VCC, introduced in Sect. 1.3, is used to verify the correctness of the annotated FreeRTOS source code, based on the function contracts derived from validated Z specification. Next, Z/Eves is a theorem prover [32], which we use in our project for analysing our Z specifications, checking syntax and proving theorems automatically with some help of human. Finally, ProZ [33] is used to animate the validated Z model. It demonstrates the behaviour of the software described by a Z specification. Sect. 2.2.1 describes Z/Eves and ProZ in detail. Furthermore, inputs and outputs of the tools are represented by circles, including documentations, FreeRTOS source code, Z specifications, etc. And the relations between the tools and their inputs and output are shown by arrows. In detail, as shown in Fig. 1.4, from the documentation [18, 19, 34] and source code of FreeRTOS, a basic version of the requirement is obtained by abstracting the documentation and reverse engineering the source code, which is described in detail in Chap. 3. Then, the first version of the Z specification is specified, based on the requirement, which is validated with the theorem prover, Z/Eves. After several rounds of iteration and improvements, the final version of validated specification is produced, which can be directly used by ProZ for animation. The validated specification and the iteration process is explained in Chap. 4. Afterwards, inspired by Multi-BSP model [35], the final specification is extended to multi-core platform and validated using Z/Eves (See Chap. 6) as well. Meanwhile, we manually translate the validated specification into VCC annotation, so, with FreeRTOS source code, the related source code can be verified (See Chap. 5).

1.5.3 Challenges

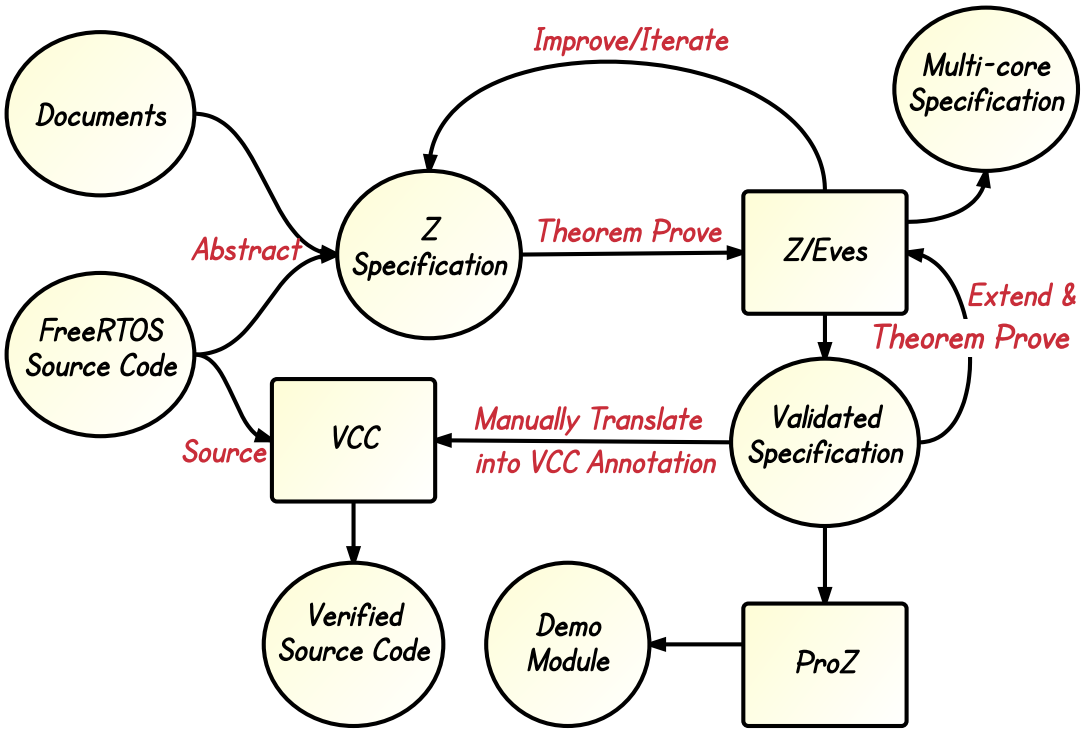
The main scientific difficulty with the verification of FreeRTOS is the low level of the code. The usual abstractions that make it easier to program systems software do not exist; it is the purpose of FreeRTOS to provide them. They include:

(a) Communication and synchronisation.

(b) Scheduling guarantees.

(c) Interference freedom.

Figure 1.4: Overview of project



(d) Direct hardware interaction using clocks and interrupts.

These are provided through a number of complex pointer-based operations, which present yet another challenge: verifying pointer programs is a complex and difficult business.

1.6 Font and Name Styles

In this thesis, text from specification and implementation are distinguished by different font styles. In detail,

- File names of the source code are represented by italic font, e.g. *task.c*;
- Teletype font is used for text from FreeRTOS source code and VCC annotation, e.g. `pxCurrentTCB`;
- For text and formulas from specification, naturally, the mathematical font is used for them, $Task$.

Table 1.1: Prefix of variable and function names used in FreeRTOS

Prefix	Meaning
x	Non-standard integer, e.g. portBASE_TYPE, portTickType, xTaskHandle, etc.
v	void return type.
ux	Unsigned non-standard integer type, e.g. unsigned portBASE_TYPE
e	Enumerated type, e.g. eTaskState
prv	Private functions, e.g. prvDeleteTCB

Table 1.2: Suffixes used in schema names

Task Model	Queue Model	Time Model	Mutex Model
$\_T$	$\_TQ$	$\_TQT$	$\_TQTM$

In addition, as there are four levels of models in the specification, each operation may have more than one version of schema for different level of the model, for instance, the operation creating task has schemas for all four models. We use suffixes in Table. 1.2 to distinguish these schemas, e.g. $CreateTask\_T$ is the schema name for creating task in the task model and $CreateTask\_TQ$ is the schema name for creating task in the queue model. Furthermore, each operation may also be composed of different sub-operations. For example, depending on whether the operation requests rescheduling, creating task has two sub-operations. We append a suffix to the end of the operation name to indicate them, e.g. N and S used respectively in $CreatTaskN\_T$ and $CreatTaskS\_T$.

Similarly, FreeRTOS uses prefixes of variable and function names to indicate the type of the variables and the return type of the functions. For instance, the prefix x shows that a variable, or the return type of a function, is a non-standard integer, e.g. portBASE_TYPE. There are two exceptions for the prefixes of the function names. Specifically, the prefix v indicates that the return type of a function is void and the prefix prv shows that a function is a private function. Table. 1.1 summarises some of the frequently used prefixes in FreeRTOS and the meaning of them. More details can be found from [36].

1.7 Structure of Thesis

The remainder of this thesis is divided into seven chapters:

Chapter 2 presents a review of the literature on formally specifying operating systems and related work in this field. It also reviews the tools used in this research.

Chapter 3 analyses the API functions of FreeRTOS and abstracts the requirements for each API function.

Chapter 4 describes the formalisation work for FreeRTOS and explains the specifications and theorems in detail. The task model shown in this chapter has been published in the journal *Formal Aspects of Computing* [37].

Chapter 5 illustrates the VCC verification for the API functions related to the task model of specifications.

Chapter 6 shows the formalisation work for the multi-core version of FreeRTOS, which is extended from the outcome of the previous chapter. The requirements are analysed and presented at the beginning. The extension to the specification for the multi-core version of FreeRTOS is then covered in the rest of the chapter.

Chapter 7 evaluates the research project in different ways. It firstly summarises the achievements of the research. It then also describes a carefully designed case study to show how the specification would work properly using the ProZ animator and the Z/Eves theorem prover.

Chapter 8 summarises the work done in this research project. Further, it reports the experience gained from undertaking the research process. Finally, suggestions for future work are presented.

Appendices introduces the structure of the following appendixes firstly. It, then, summarises the proof commands (See Page.157) frequently used in proofs of our model and the preconditions for the API functions of FreeRTOS (See Page. 159), which is followed by the specification of our FreeRTOS model and multi-core model, each of which is divided into four separate appendices according to its sub-model structure. It also presents the specification for

multi-core task model with promotion. Finally, the annotated source code related to our VCC verification is listed.

Chapter 2

LITERATURE REVIEW

This chapter reviews the literature related to the research. It begins with a detailed discussion of FreeRTOS using examples to show how the API function is used to implement various applications. Z formal notation, which is adopted in this research, is then introduced, with related tools also described. Finally, the related research on formally verifying operating systems is examined.

2.1 Related Work

This section presents the existing work on verifying operating systems:

1. Craig describes the specification and refinement in Z notation of, Labrosse's μC/OS operating system, a microkernel which is similar to FreeRTOS [38, 39, 40]. The refinement of the requirements targets mathematical data types at a level of abstraction well above program data types. The lowest level of refinement is also non-algorithmic and there are no real-time properties. Freitas & Woodcock [41] have continued Craig's refinement to target datatypes at the level of FreeRTOS, but without pointer implementation. Börger & Craig [42] also extend this work, modelling with pseudo-code descriptions as Abstract State Machines (ASMs), which produces an elegant restructuring of the model that makes it easier to understand and easier to refine into executable code.

2. Klein verifies seL4, a high-performance microkernel [43, 44, 45, 46, 47]. An abstract specification in Isabelle/HOL is refined into an executable specification in Haskell, which is then manually refined into a high performance implementation in the C programming language. The theoretical basis for the work is in separation logic. There is an almost complete handling of the features of seL4. The entire exercise involved 8,700 lines of C, 200,000+ lines of proof script, and 30 person-years of effort to establish the functional correctness of the operating system.

3. Buerki & Rueegsegger [48] introduce their design and implement a high assurance microkernel called Muen. They suggest that assuring the whole microkernel system is very difficult, but also unnecessary. However, the microkernel can be separated into different function blocks, some of which are critical. These are defined as the Trusted Computing Base (TCB) and it can lead to a fatal system error if the TCBs misbehave. Therefore, the set of TCBs can be treated as the smallest set of functions which are essential for verification.

4. Déharbe et al. have produced a specification in the B language of a restricted subset of FreeRTOS [49]. They provide a formalisation of a subset of the API function, verifying that all its expressions are well defined and demonstrate logical consistency. This model includes Task and Queue-related functions. The model contains seven basic B-machines, $FreeRTOSConfig$, $Type$, $Task$, $Queue$, $Scheduler$, $FreeRTOSBasic$, and $FreeRTOS$, with which the first model without priority is formalised. It is then refined to the second model, which takes priority into account. However, there are problems with this model; for instance, it prevents task creation while the scheduler is running, which is allowed by FreeRTOS. It also forbids tasks from sending and receiving messages to and from a queue when there is no task waiting to receive or send. Comparing this with our work, we introduce a model covering more functions of FreeRTOS. Due to a finer structure of definitions and abstractions, our specification has increased proof automation. Furthermore,

we correct the problems discovered in Déharbe et al.

5. Pronk looks at the verification problem for FreeRTOS [50]. He discusses and compares the advantages and disadvantages of theorem proving and refinement in this arena compared with model checking. He concentrates on the latter, using Promela and the SPIN model checker.

6. Lin, Freitas, & Woodcock produced a specification of FreeRTOS in Z covering the top-level functionality [51]. This was derived from Déharbe's B specification [49] (see above item 4), which was then extended to capture all the main FreeRTOS functionality. An attempt was made to verify the consistency using the Z/Eves theorem prover [32], although this could also be proved in ProofPowerZ [52], Isabelle/HOL [53], or PVS [54]. Originally, there were 30 unproven theorems out of 241. We have carried out further work to reduce the number of unproven theorems to around 10. During this process, we found the key reason for struggling with proofs is that the model is too concrete, which leads to proof complexity. For example, to represent the different states in FreeRTOS, the model uses seven different type variables, such as functions, sequences, finite sets, etc. Furthermore, even when Z/Eves can prove a theorem, it takes considerable time. Compared with this, our model is much more abstract and more tractable for proof.

7. Mühlberg & Freitas report on the application of the SOCA and VeriFast tools to FreeRTOS [55]. They focus on the verification of structural properties (e.g., pointer safety and arithmetic overflow) and liveness properties, but ultimately aim at demonstrating functional correctness. This includes the reconstruction of a formal specification of FreeRTOS in Z (mentioned above in item 6), bounded model-checking of the FreeRTOS code using the SOCAVerifier [56], as well as annotating the source code with assertions in separation logic to apply the VeriFast software verifier [57].

8. Ferreira uses separation logic to verify code-level pointer structures in FreeRTOS [58].

9. Abrial [59] has an unpublished specification of much of the functionality of FreeRTOS using the B method (excluding interrupts). The Z specification in this thesis is based on his work, although the verification is necessarily very different.

10. Mistry, Naylor, & Woodcock have developed a multi-core version of FreeRTOS on a Field Programmable Gate Array (FPGA), which is able to schedule tasks on multiple processors and support mutex in a concurrent environment [60, 61]. They present an adapted version of FreeRTOS that is able to schedule tasks on multiple processors, as well as provide full mutual exclusion support for use in concurrent applications, which is independent of the chosen platform, thus preserving one of FreeRTOS's most attractive features, portability.

11. In collaboration with the author of this thesis, an unpublished work from Kushwah, Divakaran, & D'Souza aims to give a proof of functional correctness by proving that the C implementation refines the abstract Z specification. The commonality with this work is that they also focus on the task-related functionality of FreeRTOS. The points of difference are that their specification is deterministic, more detailed, and closer to the implementation than ours. In addition, they do not check consistency or prove properties for their Z model.

12. Based on the previous work (item 11), Divakaran et al. [62] also attempt to use VCC to verify the implementation of FreeRTOS with abstract Z specification. The difference between these works is that they focus on checking refinement steps. They promote three approaches (*"Direct-Import"*, *"Combined"* and *"Two-Step"*) in VCC to check refinements between the abstract model and the implementation. By comparing and applying them to the case study, they claim that the *"Two-Step"* approach is much more efficient than the other two, which splits the verification process into two steps: (a) The behaviours of the function defined in the abstract model and the concrete function are verified; (b) Then, the outputs from both functions satisfying the

gluing invariants is checked. Furthermore, they also suggest how to translate a subset of the Z notation to VCC.

2.2 Z Notation

Z, developed by Sørensen [63] in 1982, is a formal notation and became an international standard in 2002 [64]. It is based on set theory and mathematical logic. Specifically, the Z notation uses set operators, set comprehensions, Cartesian products and power sets. The logic part uses first-order predicate calculus [65]. With these mathematical theories, Z can describe the state and properties of the system being specified. Z uses an abstract structure called a Schema to capture a number of concepts in one named block. Subsequent schemas that refer to the name of a previous schema can inherit all the concepts and constraints introduced in that schema [10, 63, 65, 66]. This provides reusability in the Z notation. In order to manage system complexity, schemas are vital to keep the specification flexible and manageable in a real, large-scale project [65]. Moreover, a schema can also be used to specify the behaviour of the system.

Hoare [67] introduced Hoare Triples in 1969, which describe the logical behaviour of a computer program. This triple can later be verified by related logic theories and lemmas. Specifically, the notation

$$P\ \{Q\}\ R$$

can be used to express that if the precondition P is true before the program Q is initialised, then the postcondition R will be true once Q terminates successfully. Therefore, a schema is composed of two parts, a precondition and a postcondition. The default relation between predicates in different lines of a schema is called logical conjunction $\wedge$.

For instance, the following shows part of a specification for a system. It is used to record the relationship between tasks and their priority. (Note, this is just a example to illustrate how schemas work; the definition here is not the same in our specification of FreeRTOS.) Firstly, $TASK$ is defined to represent tasks in the world using a given set, which is used to introduce uninterpreted domain-specific types in Z.

$[TASK]$

Subsequently, a schema, $Task$, is introduced to specify the basic abstract state of the system. It includes two component declarations: (a) $tasks$, which is a subset of $TASK$, indicates the tasks in our system; and (b) $priority$, which is a partial function from $TASK$ to $\mathbb{N}$, illustrates that tasks may have a priority. As well as the declaration, the constraint for these properties is also defined, indicating that the domain of $priority$ is $tasks$. This means tasks in set $tasks$ have a priority and tasks not in $tasks$ do not.

Task

$$tasks : \mathbb{P}\ TASK$$
$$priority : TASK \pfun \mathbb{N}$$

$$\mathrm{dom}\ priority = tasks$$

Based on state schemas, operation schemas can be defined, such as $setPrio$ below, which specifies the operation for setting the priority of a task. It refers to the $Task$ schema, to obtain all the essential information about the system we defined. Therefore, the variables defined in $Task$ can be directly used.

setPrio

$$\Delta Task$$
$$t? : TASK$$
$$prio? : \mathbb{N}$$
$$out! : \mathbb{N}$$

$$t? \in tasks$$

$$tasks' = tasks$$
$$priority' = priority \oplus \{(t? \mapsto prio?)\}$$
$$out! = priority'(t?)$$

The declaration part of the schema starts with $\Delta Task$, which refers to the pre- and post-state of the $Task$ schema and indicates that it is going to change the state $Task$. Following this, it introduces some inputs (e.g., $t?$) and outputs (e.g., $out!$). They are distinguished through a decoration convention. Variables in the pre-state (e.g., $tasks$) and post-state (e.g., $tasks'$) are similarly distinguished. This schema has only one precondition, $t? \in tasks$, indicating that before the operation

$setPrio$ the target task $t?$ is in the system. Therefore, it is in the domain of $priority$ as defined in $Task$ schema. Consequently, after the operation has successfully finished, the new priority of $t?$ is updated in the function $priority'$. The override operator, $\oplus$, is used to achieve this, which is the most common way to update functions in the Z notation. If the first element of the pair specified in operation (e.g., $t?$ in this case) exists in the domain of the function (e.g., $priority$), it would update the result of function application (e.g., $priority(t?)$) to the new value (e.g., $prio?$); otherwise it adds the pair into the function. Finally, the new priority of target task $t?$ is set as output, which should simply equal to $prio?$.

More explanation of the Z notation will be provided in Chap. 4 along with the explanation of our specification.

2.2.1 Tools for Z

A leading proof tool for the Z notation is Z/Eves, which can be used for analysing Z specifications, syntax checking and most importantly theorem proving. Using Z/Eves, specifications can be entered by importing LaTeX source code [68] or by typing directly from the editing facility provided by Z/Eves [69]. Once the specification has been entered, Z/Eves can automatically perform syntax checking, type-checking, and some basic proving, by double clicking on paragraphs, such as schemas. This is very useful for users to avoid syntax errors. Furthermore, based on self-defined theorems or lemmas and the built-in theorems, which can be found in [70], Z/Eves can be used to prove specifications automatically. More details about how to use Z/Eves can be found in the user guide [69].

ProZ, which is extended from ProB [33], can be used to animate and check Z specifications. It is worth noting that each specification animated by ProZ should have one and only one schema named Init, which defines the initialisation state of the model. Once the model is initialised, based on the current state of the model, available operations will be shown and the user can apply them to the model by simply double clicking on them.

2.3 FreeRTOS

FreeRTOS is a widely used real-time operating system written by a team led by Richard Barry of Wittenstein High-Integrity Systems in the UK [19]. Introductions to FreeRTOS informally describe the application programming interface (API) for the real-time operating system kernel [18]. Verifying the correctness of FreeRTOS has also been proposed as a pilot project for the VSI. This verification experiment presents two distinct challenges: (a) Code-level verification to automatically analyse FreeRTOS for structural integrity properties; and (b) The creation of a rational reconstruction of the refinement of the FreeRTOS code starting from an abstract specification, discharging all verification conditions automatically. This project was chosen as a contribution to the VSI at a workshop held at Microsoft Research, Cambridge, in 2008, that was gathering difficult research problems from industry. Modelling and verifying operating system kernels is considered to be scientifically interesting, pushing the current capabilities of software verification research and technology. Klein is the first to formally verify an operating system kernel and describe the main scientific challenges [43, 71].

FreeRTOS has a large community of users programming embedded microcontrollers: it was downloaded 107,000 times in 2013, putting it high in the top 100 SourceForge codes (there are more than 200,000 available). Verification of FreeRTOS, which allowed the discovery of residual errors, would thus have a strong impact on the international embedded system community.

FreeRTOS is a lightweight, embeddable, multi-tasking, Real-Time Operating System (RTOS). It makes the key assumption that the target system has a single processing unit. It is really a library of types and functions that can be used to build microkernels using a combination of C and assembly language, and has been ported to most embedded systems architectures. It allows a very small kernel to be produced to target microcontrollers, somewhere between 4–9kB. In some special cases, it can be less than 4kB. For instance, it takes less than 4kB of RAM, when creating 13 tasks, 2 queues and 4 software timers for RL78 [34]. It provides

services for embedded programming tasks, communication and synchronisation, memory management, real-time events and I/O-device control.

Fourteen different compilers are used with FreeRTOS, giving complex configuration options and extensive parametrisation. A version of the software, SafeRTOS, has been certified to Safety Integrity Level 3 by the Technical University of Vienna for the following safety standards: IEC 61508, FDA 510(k), and DO-178B. These certificates are for the process of development, rather than for the correctness of the software against stated requirements.

The objective of formally verifying FreeRTOS would be to find any errors and make some guarantees about the code's behaviour. Since the requirements are distributed throughout the documentation, there is a clear need to produce a formal abstract specification. A broader aim of our work is to study the verification problem for an entire class of software, namely *real-time operating systems for embedded applications*, and we have chosen to focus on an exemplar of this class of system namely the FreeRTOS kernel. The techniques and methodology developed here can be expected to be applicable to other software in this class of system.

2.4 Summary

This chapter firstly reviewed previous work related to formalising operating systems, which is helpful for us to understand the background of the project. Secondly, it discussed the principle of formal verification and Z used in the project. In addition, it reviewed the target system of the project, FreeRTOS.

Based on this, the next chapter will define the goals and scope of the project and most importantly the requirements of the project can be abstracted.

Chapter 3

Analysis and Abstract API functions of FreeRTOS

This chapter discusses the requirements for the research. Because the development process does not completely follow either a formal methods strategy or reverse engineering strategy, it begins by giving an overview of our model. It then discusses the goals and scope of the research, followed by the requirements analysis.

3.1 Model Overview

Normally, the development process in formal methods starts with the requirements, modelling the behaviour of the system, and refining through several steps to executable code. As FreeRTOS does not have explicitly articulated requirements and has been implemented in C, we consider the user manual and the practical guide to FreeRTOS [19, 18] the basis of the requirements. However, these sources are not detailed enough for us to build the model; they just provide a basic functional description of the API functions. Therefore, we also take the FreeRTOS source code into consideration for modelling. Thus, the model is mainly based on the API documentation to verify the functional correctness of FreeRTOS, with some of the details of the specification derived from the source code. For instance, the

`xTaskCreate` API function documentation states that if the API function returns $pdTRUE$, then the task has been created. (NB, to simplify the model, we just consider the successful case of API functions.) Nevertheless, it does not indicate how it was created. Therefore, we analysed the source code to find out how it works and formalised the behaviour of `xTaskCreate` based on that.

As described in the last chapter, the key elements of FreeRTOS can be divided into three categories: (a) task management; (b) communication and synchronisation; and (c) memory and interrupt management. However, functions related to memory and interrupt management are quite hardware-dependent, so we abandon them at this level of abstraction. Furthermore, according to whether they are time dependent, task API functions can be divided into pure task operations or time-related operations. Therefore, we focus on task, communication (i.e., queue), time and synchronisation (i.e., semaphore and mutex) related FreeRTOS API functions. Each of them is reasonably independent. Therefore, our modelling starts from the core part of the system, $Task$ model, and then expands to cover other features. For each subsystem, we attempt to keep the model as simple as possible. This can significantly reduce the difficulty of modelling and verification. In detail, we first build the $Task$ model. Based on this, the $Queue$ model is added into the system, followed by the $Time$ and $Mutex$ models.

3.2 Goal and Scope

The goal of this research is to provide a verified high level abstract formal model for FreeRTOS, which can be used as the foundation for future research (i.e., refinement, extension, etc.). It describes the behaviour of FreeRTOS API functions. Based on FreeRTOS's manual [19], 61 API functions are provided to the user. However, some of them have similar functions, for instance, $xQueueReceive$ and $xQueuePeek$. Both of them attempt to receive an item from a queue, but $xQueueReceive$ actually receives data from the queue and $xQueuePeek$ just checks if there are any items available in the queue. Moreover, due to the abstract level of the specification, some API functions perform the same function, for instance,

$xQueueSend$, $xQueueSendToBack$, $xQueueSendToFront$. All of these try to send an item to a queue. The difference is that with $xQueueSend$ a user can send the item to either the front or the back of the queue and the latter two API functions can only send the item to the back and the front of the queue respectively. At this level of abstraction, we do not need to consider the detail of the order of items in a queue or how items are stored in a queue. Therefore, we only selected 15 of the API functions for our model (listed in Sect. 3.3.1). To simplify the modelling and verifying further, we assume that the system scheduler is continuously running and only focus on cases where the API functions succeed.

Due to the complexity of the traditional refinement process and time limitations, we have not refined our model into executable code. However, we attempt to use VCC and our model to verify task-related FreeRTOS API functions to illustrate that the VCC kind of verifier plus abstract specification can be an alternative approach to verification. In addition, we also extend this model for multi-core platforms.

3.3 Requirements

Generally, the use case diagram, which shows the relation between the actor and use cases, and the relation between different use cases, is the most common approach to describing requirements. It is also used as a fundamental document for further software development. However, in our project, FreeRTOS has been implemented already and does not have a proper set of requirements. Therefore, we summarise requirements for FreeRTOS according to its API function.

3.3.1 Functional Requirements

1. **Task Related:**

    1.1 $xTaskCreate$ Create a task and specify its priority, reschedule tasks when the priority of the new task is greater than the running task;

    1.2 $vTaskDelete$ Remove a non-idle task from the system, reschedule tasks if the deleted task is the running task;

1.3 $vTaskSuspend$ Set the state of a task to suspended, reschedule tasks if the suspended task is the running task;

1.4 $vTaskResume$ Set the state of a suspended task to ready, reschedule when the priority of the resumed task is greater than that of the running task;

1.5 $vTaskPrioritySet$ Set new priority of a task (N.B. the priority of `idle` task is always 0), reschedule tasks:

- if the priority of the running task is set lower than the priority of the highest-priority ready task;
- if the priority of a ready task is set greater than that of the running task.

2. **Queue Related:**

2.1 $xQueueCreate$ Create a queue and specify its size;

2.2 $vQueueDelete$ Delete a queue;

2.3 $xQueueSend$ Send an item to the queue;

- block this task if the queue is full, add it to waiting send list;
- if the queue is empty, wake up the highest-priority task, which is waiting to receive an item from the queue.

2.4 $xQueueReceive$ Receive an item from the queue;

- if the queue is empty block the task and add it to waiting receive list;
- if the queue is full, wake up the highest-priority task, which is waiting to send an item to the queue.

3. **Time Related:**

3.1 $vTaskDelayUntil$ Block current running task until the specified time, reschedule the highest-priority ready task as the new running task;

4. **Semaphore & Mutex Related:**

4.1 $vSemaphoreCreateBinary$ Create a binary semaphore;

4.2 $vSemaphoreDelete$ Delete a binary semaphore or a mutex;

4.3 $xSemaphoreCreateMutex$ Create a mutex;

4.4 $xSemaphoreTake$ Take a token from a semaphore or a mutex;

4.5 $xSemaphoreGive$ Give a token back to a semaphore or mutex;

In addition, the following two time properties are related to task scheduling, which are also of interest to us:

1. The function $prvCheckDelayedTasks$ checks expiry time for blocked tasks. When time increases, it checks if there are any blocked tasks that need to be woken up. If there are, it moves them to the ready state, which may cause rescheduling;
2. When the system increases the ticks counter, which represents the time, it also checks whether there are any ready tasks with the same priority as the running task to share the processor.
   - If there is more than one ready task sharing the highest priority, they need to share CPU time as well. Rescheduling is required in this case.
   - Otherwise, the current running task keeps running.

3.3.2 Non-functional Requirements

Non-functional requirements specify the constraints on the services or functions offered by the system. In our case, the non-functional requirements can be summarised as:

1. Well-definedness. The specification should be well-defined;
2. Animatable. The specification should be able to be animated by ProZ.
3. Feasibility. The specification should be feasible (i.e. initial state and precondition for each operation should be reachable);
4. Reproducible. The specification and verification should be easily reproduced by other users.
5. Reusable. The specification should be able to be reused and expanded easily with little or no modification;

3.3.3 Environment Requirements

Z/Eves and ProZ are used as the prover and animator during the project. To reproduce the experiment, they are essential. Additionally, Community Z Tools (CZT) [72] can be very helpful for modifying the specification source code. CZT also integrates an interface for Z/Eves, as the original graphic interface of Z/Eves is implemented in Python, which crashes easily in Windows 7. Further, FreeRTOS v7.1.1 is used for the project.

3.4 Summary

In this chapter, the first section showed how we abstract the requirement for FreeRTOS, following which the second section defined the goal and scope of the project. Finally, the last section discussed the requirements of the project in three parts: functional requirements, non-functional requirements and environment requirements. Specifically, the functional requirements described the requirements for FreeRTOS API functions we modelled in the project, which were divided into four categories.

In the next chapter, following the categories of the functional requirements, the abstract model of FreeRTOS, which is the core of the project, will be described in detail. In addition, the experience gained from the modelling process will be discussed, which is helpful for developers specifying large systems, like us.

Chapter 4

MODELLING FREERTOS

This chapter describes the model in detail, following the structure described in the previous chapter. Firstly, it shows how we approached the model. The $Task$ model is then described, being the simplest and most important part of the model. In this section, some auxiliary theorems, which are helpful during proving and modelling are also explained. Following this, the $Queue$ model is illustrated, followed by the $Time$ and $Mutex$ models. For each of these models, we also briefly explain how the API functions of the previous model are expanded. In addition, we collect the preconditions for each API function and some properties of the system. We also give a summary of the proof commands in Appendix B, so that the reader can follow the general argument behind the formal proofs or even recreate the proof in Z/Eves.[1]

4.1 Iteration Process

Following the requirement and the refinement strategy of [59], we started the modelling process with the simplest and the most important part of the system, which includes creating tasks, deleting tasks and rescheduling. Basically, the idea of creating a task is adding a task, which does not belong to the system, to the system. To describe this, besides the given set ($TASK$) representing tasks, we need

[1] Z/Eves project file and other related files can be found in supplementary material.

a set ($tasks$) to help us to distinguish the tasks known by the system and others. Similarly, to delete a task, we simply remove the task from the system, i.e. remove the target task from the set $tasks$. Finally, to define the behaviour of rescheduling, the simplest idea is setting the target task as the new running task, and performing a context switch. The variables, $running\_task$, $log\_context$ and $phy\_context$ are defined for this purpose. Therefore, we specified the first and tiniest model of FreeRTOS, which contains only one base schema called $FreeRTOS$.

$$
\begin{array}{|l}
\hline
FreeRTOS \\
\hline
phy\_context : CONTEXT \\
running\_task : TASK \\
log\_context : TASK \rightarrow CONTEXT \\
tasks : \mathbb{P}\, TASK \\
\hline
running\_task \in tasks \\
idle \in tasks \\
\hline
\end{array}
$$

The schema $FreeRTOS$ contains all the necessary variables described above to describe creating, deleting and rescheduling tasks. Based on this, we defined the specification for these three operations. As this model is really tiny and simple, it is easy to understand and validate, but extremely incomplete. We then took some fundamental attributes of tasks into consideration. First, a task in FreeRTOS always has a state and it should be possible to change the state of a task from one to another. Second, the scheduling policy adopted by FreeRTOS is priority-based scheduling. A task in FreeRTOS must have a priority. The task then can be scheduled according to its priority. Therefore, the functions, $state$ and $priority$ are added into $FreeRTOS$. The specification for creating, deleting and rescheduling tasks are updated accordingly. Meanwhile, due to these two new functions, we can define the behaviour of suspending, resuming and changing priority of tasks. At this stage, we obtained a reasonable complete task model of FreeRTOS. As the model is still simple (compared to the expanded models) and we did not verify preconditions for each schema, there were no issues during the modelling and proving process. Identifying system attributes, encapsulating the attributes to a base schema and specifying the related behaviour of the system based on the schema can be considered as a good choice for a small system. Because of the

simplicity of the system, the size of base schema can be reasonably small. In this case, encapsulating all related attributes in the base schema does not raise the difficulty and complexity of proof for other schemas in the model.

After this, following the structure described in Chap. 3, we extended the model incrementally to contain queues, then added time, and finally added semaphores and mutexes. Similarly, each time, all base information, which will be introduced in detail in Sect. 4.3, 4.4 and 4.5, was added to the base schema $FreeRTOS$ and the related operations are defined based on $FreeRTOS$. We found that during this process we produced an unacceptably large base schema. At the end, when semaphores and mutex related data were added to the model, the base schema $FreeRTOS$ was longer then an A4 page. As a consequence, operation schemas also became unreasonably long, which increased the difficulty of validation and made the preconditions for operation schemas impossible to verify. The successful approach described above for the task model becomes unsuccessful, as the scale of the system increases the difficulty and complexity of the model dramatically.

To solve this problem, we broke the base schema, $FreeRTOS$, down into smaller pieces. As stated in the previous chapter, the base schema is split into four sub-schemas, $Task$, $Queue$, $Time$ and $Mutex$. They are the four sub-models which are described in following sections. Using this strategy, the size and complexity of the schemas of the model are reduced dramatically. However, these sub-schemas still contain too much information to verify the preconditions for operation schemas, which produce too much unrelated information in the proof condition. This trivial information increases complexity and the difficulty of auto proving for the prover. Thus, the sub-base schemas were broken down further to obtain the current version of the model. Based on our experience, we can state that encapsulating all the attributes of the system to a single base schema is definitely an unsuccessful approach for a large system; on the other hand, hierarchically structured incremental base schema can be considered as a good choice, as it reduces the difficulty and complexity of the modelling and proving process by: (a) hiding as much unrelated information as possible, (b) proving theorems in the model which contains only the

information related to the theorem, (c) reusing the previous proved theorems for proving theorems in later complex model.

4.2 Task Model

As mentioned in Sect. 3.1, task related API functions are the core part of the model and fundamental to it. To define this model, it is essential to state some basic context that will be used in the specification.

4.2.1 Basic Statements

The given sets $CONTEXT$ and $TASK$ are provided as given sets to represent the environment of the processor and the tasks, respectively; in Z, given sets are basic, maximal types.

$$[CONTEXT, TASK]$$

Two constants, $bare\_context$ and $idle$, are introduced by an axiomatic definition, which contains a declaration and a constraint. Here, the constraint is trivially true and is omitted. The constant $bare\_context$ is an element of the set $CONTEXT$; it represents the initial state of the processor. The constant $idle$ is of type $TASK$; it represents the system task that runs when no other task is scheduled.

$$\begin{array}{|l} bare\_context : CONTEXT \\ idle : TASK \end{array}$$

$STATE$ is defined using a free type in its simplest form, enumerating exactly five distinct constants.

$$STATE ::= nonexistent \mid ready \mid blocked \mid suspended \mid running$$

The set of legal state transitions is described by an abbreviation: $transition$ names the appropriate set that models the diagram in Fig. 1.1.

$$\begin{aligned} transition == & (\{blocked\} \times \{nonexistent, ready, running, suspended\}) \\ \cup & \ (\{nonexistent\} \times \{ready, running\}) \\ \cup & \ (\{ready\} \times \{nonexistent, running, suspended\}) \\ \cup & \ (\{running\} \times \{blocked, nonexistent, ready, suspended\}) \\ \cup & \ (\{suspended\} \times \{nonexistent, ready, running\}) \end{aligned}$$

In particular, transitions $(blocked, running)$ and $(suspended, running)$ are included because when a task is woken up from the `blocked` state or resumed from the `suspended` state, its state actually transits to `ready`. However, if it has a higher priority than the running task, it will be scheduled to `running`. At this level of abstraction, we consider these two steps as a single step, which makes state transitions $(blocked, running)$ and $(suspended, running)$ possible. The definition for $transition$ turns out not to be very useful in automating proofs about transitions, because Z/Eves would expand $transition$ into the set in all possible proof contexts. This greatly increases the load on the prover. Therefore, we `disable` the definition and add two theorems that are more helpful. The first is a typing lemma that states that $transition$ is a set of pairs of $STATE$; its proof is a very simple consequence of the definition of $transition$. With the help of the proof command prefix "$with\ enabled\ (transition)$", Z/Eves will take the disabled definition of $transition$ into consideration during proof. The proof command "$prove\ by\ reduce$" requests Z/Eves to explore possible theorems and lemmas to prove the goal automatically. Therefore, the goal can be easily proved automatically by Z/Eves using the following command.

Theorem 1 (gTransitionType)

$$transition \in \mathbb{P}(STATE \times STATE)$$

proof $[gTransitionType]$
$with\ enabled\ (transition)\ prove\ by\ reduce;$
■

Next, we add the following lemma to tell Z/Eves about each individual pair in $transition$, which is helpful to Z/Eves for automatically proving. Similarly, the proof is very simple.

Theorem 2 (rule IInTransition)

$$\begin{array}{l} \forall\, l, r : STATE \\ \quad \mid (l, r) \in \{(nonexistent \mapsto ready), (running \mapsto ready), \\ \qquad (blocked \mapsto ready), (suspended \mapsto ready), \\ \qquad (ready \mapsto running), (blocked \mapsto running), \end{array}$$

$(suspended \mapsto running), (nonexistent \mapsto running),$
$(running \mapsto suspended), (ready \mapsto suspended),$
$(blocked \mapsto suspended), (running \mapsto blocked),$
$(running \mapsto nonexistent), (ready \mapsto nonexistent),$
$(blocked \mapsto nonexistent), (suspended \mapsto nonexistent)\}$

- $(l, r) \in transition$

proof $[lInTransition]$
with normalization with enabled $(transition)$ *prove by reduce*;
■

Based on these definitions, the state schema of the model can be specified, describing basic system properties. For this stage of modelling, we focus only on task-related information in FreeRTOS. Further information will be introduced in a related model. To simplify the proof and the specification, we verify the system only when the scheduler is running. Therefore, we assume the scheduler is always running.

To describe the tasks in FreeRTOS, the following four kinds of data are needed, which are defined by a schema definition. In the Z notation, the schema is used to structure and compose descriptions. Once a schema is assigned a name, it is possible to use that name to reuse the schema in other expressions or schemas.

1. **Task data.** The variables recorded in this category are directly related to tasks. First, to simplify the description of the model and the following proofs, we need to distinguish tasks that are known to the system from others; therefore, a set $tasks$ is defined as a finite subset of $TASK$. Second, in the FreeRTOS source code, $task.c$ file, a pointer (`pxCurrentTCB`) is used to record the current running task, which is useful in several cases, such as scheduling. In the specification, a variable $running\_task$ of type $TASK$ is used to represent this. Two constraints are specified: the $idle$ task and the $running\_task$ have to be known to the system at all times.

$TaskData$
$tasks : \mathbb{F}\ TASK$
$running\_task : TASK$

$$running\_task \in tasks$$
$$idle \in tasks$$

2. **State data.** As described in Sect. 1.2.1, FreeRTOS uses different lists to manage the tasks known to the system. Abstractly, two tasks in different lists have different states. Therefore, the variable $state$ is used to indicate the state of the tasks. Specifically, the $idle$ task, which is a system task with responsibility for maintenance jobs for the system (such as garbage collection), can only be $ready$ or $running$; it cannot be $blocked$, $suspended$, or deleted ($nonexistent$).

$StateData$
$$state : TASK \rightarrow STATE$$
$$state(idle) \in \{ready, running\}$$

3. **Context data.** The two variables $phys\_context$ and $log\_context$, respectively, represent the physical system context (e.g., register values, some stacks, *etc.*) and the logical context for all the tasks that are not running (i.e., the system states of a task when it exits the running state).

$ContextData$
$$phys\_context : CONTEXT$$
$$log\_context : TASK \rightarrow CONTEXT$$

4. **Priority data.** FreeRTOS is a priority-based operating system: all the tasks in the system have their own priority, and a total function, $priority$, is introduced to record this. The priority of $idle$ task must always be the lowest priority, which is 0.

$PrioData$
$$priority : TASK \rightarrow \mathbb{N}$$
$$priority(idle) = 0$$

Invariant Based on these definitions, we can describe the state schema for tasks that are maintained by this part of FreeRTOS.

$$
\begin{array}{|l}
\hline
Task \\
\hline
TaskData \\
StateData \\
ContextData \\
PrioData \\
\hline
tasks = TASK \setminus (state^{\sim}(\!| \{nonexistent\} |\!)) \\
state^{\sim}(\!| \{running\} |\!) = \{running\_task\} \\
\forall\, pt : state^{\sim}(\!| \{ready\} |\!) \bullet priority(running\_task) \geq priority(pt) \\
\hline
\end{array}
$$

Apart from the four schemas describing the task, state, context, and priority data, three more constraints are added to this schema. They show that:

- All the tasks whose state is not $nonexistent$ are known to the system. Here, as mentioned above, the $state$ is a function, a special case of a relation. The operator, $^{\sim}$, takes the inverse relation, so that $state^{\sim}$ is a relation in $STATE \leftrightarrow TASK$. The operand, $(\!|$ and $|\!)$ calculates relational image. The result for this predicate is a set that contains all the $TASK$s whose states are $nonexistent$.

- Only one task can occupy the $running$ state at any given time, which is $running\_task$.

- The priority of the $running\_task$ is the highest of all the ready tasks.

Initialisation Based on the state definition and the assumptions mentioned above, we describe the initialisation of the $Task$ state in a similar piecewise fashion: we separately initialise the four sub-states, and then combine them.

1. **Task data.** Initially, there are no user-defined tasks in the system; there is only one task in the system, $idle$, which is also the initial $running\_task$.

$$
\begin{array}{|l}
\hline
Init\_TaskData \\
\hline
TaskData' \\
\end{array}
$$

$$tasks' = \{idle\}$$
$$running\_task' = idle$$

2. **State data.** Furthermore, every other task is in the $nonexistent$ state, except $idle$ whose state is $running$.

$Init\_StateData$

$$StateData'$$

$$state' = (\lambda\, x : TASK \bullet nonexistent) \oplus \{(idle \mapsto running)\}$$

3. **Context data.** Also, initially, the logical and physical contexts of all tasks is the $bare\_context$.

$Init\_ContextData$

$$ContextData'$$

$$phys\_context' = bare\_context$$
$$log\_context' = (\lambda\, x : TASK \bullet bare\_context)$$

4. **Priority data.** Finally, all tasks have the lowest priority, 0.

$Init\_PrioData$

$$PrioData'$$

$$priority' = (\lambda\, x : TASK \bullet 0)$$

The initial state for $Task$ can be defined using these four definitions.

$Init\_Task$

$$Task'$$

$$Init\_TaskData$$
$$Init\_StateData$$
$$Init\_ContextData$$
$$Init\_PrioData$$

In order to prove that all the initial states are reachable, the following five theorems are introduced. They assert that there is at least one possible postcondition for initialising each sub-state schema and the overall schema. Due to the simplicity of these theorems, Z/Eves is able to fully prove them automatically.

Theorem 3 (TaskDataInit)

$$\exists\, TaskData' \bullet Init\_TaskData$$

proof [$TaskDataInit$]
$prove\ by\ reduce$;
■

Theorem 4 (StateDataInit)

$$\exists\, StateData' \bullet Init\_StateData$$

Theorem 5 (ContextDataInit)

$$\exists\, ContextData' \bullet Init\_ContextData$$

Theorem 6 (PrioDataInit)

$$\exists\, PrioData' \bullet Init\_PrioData$$

It is easy to prove these theorems with the proof command "$prove\ by\ reduce$", except for $TaskInit$, because it has more constraints on its state variables.

Theorem 7 (TaskInit)

$$\exists\, Task' \bullet Init\_Task$$

proof [$TaskInit$]
$prove\ by\ reduce$;
$apply\ extensionality$;
$with\ enabled\ (applyOverride)\ prove$;
■

After the automatic proving ordered by $prove\ by\ reduce$, Z/Eves is confused about the equivalence between sets defined in schema $Task$. The application of override also confuses the prover. Therefore, we need to guide the prover to apply theorems, $extensionality$ and $applyOverride$, to discharge them. These theorems are provided by the Z/Eves toolkit [70].

We can check whether the state change respects the transition relation as a dynamic invariant that must be satisfied by all the operations on the $Task$ state by redefining $\Delta Task$:

$\Delta Task$

$Task$
$Task'$

$\forall\, st : TASK \mid state'(st) \neq state(st)$
$\quad \bullet\ state(st) \mapsto state'(st) \in transition$

It is worth mentioning that in this schema we use $Task'$ to refer to the post state of the $Task$. Initially, the expression "$\Delta Schema$" ($Schema$ refers to a state schema) has been defined to contain both the pre- and post-state of $Schema$. We redefine it here to add further constraints for $Task$.

Based on these fundamental definitions, operations related to tasks can be specified.

4.2.2 Additional Schema for Reschedule

In a multi-tasking real-time operating system, rescheduling tasks is essential and occurs frequently. Generally, depending on the purpose of the system, the operating system would follow some suitable algorithm to determine the task to be scheduled. Other system states can then be updated accordingly. Therefore, at this level of abstract specification, it is possible to define the rescheduling process nondeterministically. However, the model described in this chapter focuses on FreeRTOS. We will follow the algorithm used in FreeRTOS to specify rescheduling, which is based on task priority. Specifically, once a `ready` task obtains a higher priority than the running task, it will be scheduled as the new running task. Subsequently, the system will switch the context of the current running task out and swap in the context of the new running task. It is also necessary to manage related lists and system states properly, for instance, by setting the selected task as the running task and inserting the current running task in a suitable list.

In this specification, we introduce the schema $Reschedule$ to perform the swapping part of the rescheduling process, which can then be used by other schemas.

The priority-based scheduling algorithm is embedded in the operation schemas for different API functions that need rescheduling. The priority-based rescheduling behaviour depends on the destination to which the current running task is moved. For example, when suspending the running task, the destination of the running task is the suspended list; but when we create a task with a higher priority than the running task, the destination of the current running task is actually one of the ready lists. These lists are represented by the function $state$. Therefore, updating the $state$ with the variable $st?$ manages these lists. In the Z notation, variables marked with "$?$" and "!" indicate that they are I/O variables, respectively, for a schema. When other schemas reuse the $Reschedule$ schema, $st?$ will be introduced within these schemas with the value of the destination of the current running task. Because both schemas contain a variable with the same name, these two variables will be bound together. Consequently, the schema $Reschedule$ can obtain the destination of the running task by accessing the value of $st?$. The operator, $\oplus$, is normally used to update functions in Z. If the first element of a pair exists in the domain of the function, it will update the second element of the pair in the function to the new value; otherwise it appends the pair to the function. Therefore, it is used here to update the $state$ of $running\_task$ and $target?$. Similarly, for each case, the new running task, the final state of $tasks$, and the priority of tasks may also be different. We leave these decisions to the calling schemas. Therefore, variables—$target?$, $tasks?$ and $pri?$—are introduced to represent these properties.

$$
\begin{array}{|l}
\hline
Reschedule \\
\Delta Task \\
target? : TASK \\
tasks? : \mathbb{P}\, TASK \\
st? : STATE \\
pri? : TASK \to \mathbb{N} \\
\hline
tasks' = tasks? \\
running\_task' = target? \\
state' = state \oplus \{(target? \mapsto running), (running\_task \mapsto st?)\} \\
phys\_context' = log\_context(target?) \\
log\_context' = log\_context \oplus \{(running\_task \mapsto phys\_context)\} \\
priority' = pri? \\
\hline
\end{array}
$$

The calling schema just needs to specify the correct values for these variables, the $Reschedule$ schema then handles the rest of the work.

4.2.3 Creating and Deleting Tasks

After initialising the system, there is only one task ($idle$); in order to add more tasks to the system, the $Create$ operation can be used. Once a task finishes, it should be $Delete$d (see Page. 55) to allow other tasks to use the resources held by it. `xTaskCreate` and `vTaskDelete` are also the first group of API functions provided by FreeRTOS. Generally, there are two cases for each of these two operations: one is to add or remove a task from the system; the other one leads to a re-scheduling of tasks.

First Case of Creating Tasks If the assigned priority is not greater than the priority of the current running task, it adds the new task that does not already exist. The input $target?$ represents the task that will be created. The input $newpri?$ contains the priority assigned to the new task. Therefore, the precondition is specified as: first, $target?$ is not known by the system; second, the assigned priority, $newpri?$ is no more than the priority of $running\_task$. After the operation, the $target?$ is known to the system, the task $target?$ is added to $tasks$ and updates the $state$ function to record that the state of $target?$ is $ready$. The input $newpri?$ is assigned to the task $target?$ by updating the function $priority$. Because this operation will not cause rescheduling, other properties of $Task$ remain unchanged. The "Ξ" operation has been used here: it is defined in Z to show that the pre- and post-states are unchanged. The schema $CreateTaskN\_T$ can be introduced, which indicates that this schema is used to *Create Task operation* in the *normal case* for the *Task model*. Generally, we use postfix N for the *Normal case* of the operation, which does not lead to rescheduling; and S for the *Scheduling case*. The postfix after the underscore indicates which model it is specified for. For example, *T* in this case shows the schema is part of the task model.

$CreateTaskN\_T$

$\Delta Task$

$$
\begin{array}{|l}
target? : TASK \\
newpri? : \mathbb{N} \\
\hline
state(target?) = nonexistent \\
newpri? \leq priority(running\_task) \\
tasks' = tasks \cup \{target?\} \\
running\_task' = running\_task \\
state' = state \oplus \{(target? \mapsto ready)\} \\
\Xi ContextData \\
priority' = priority \oplus \{(target? \mapsto newpri?)\} \\
\hline
\end{array}
$$

Having defined this operation as a relation on $Task$ states, we need to work out its precondition. We posit that the before-state, the inputs, and the first two predicates are the precondition, and collect these into the following schema, where the suffix $FSBSig$ in the schema name stands for *Feasibility Signature*.

$$
\begin{array}{|l}
\hline
CreateTaskN\_TFSBSig \\
\hline
Task \\
target? : TASK \\
newpri? : \mathbb{N} \\
\hline
state(target?) = nonexistent \\
newpri? \leq priority(running\_task) \\
\hline
\end{array}
$$

These declarations and predicates are clearly necessary for the actual precondition as stated above. We show that they are also sufficient in the next theorem, which can be automatically generated. Specifically, for any "state" that satisfies the definition of $CreateTaskN\_TFSBSig$, the precondition of $CreateTaskN\_T$ is satisfied. The operator "$\mathrm{pre}\ Schema$" is defined in Z to calculate the precondition schema of a schema [65, Chap. 14]. For instance, the predicate "$\mathrm{pre}\ CreateTaskN\_T$" in the following theorem obtains the precondition schema by calculating $\exists\, Task' \bullet CreateTaskN\_T \setminus (outputs)$, where $outputs$ refers to the list of output variables related to the operation, which will be hidden, and is empty in this case. The schema hiding operator, "$\setminus$", hides the variables listed in the $outputs$ from the declaration of the operation by introducing them in the predicate part of the schema with an existential quantifier.

Theorem 8 (CreateTaskN_T_vc_ref)

$$\forall\, CreateTaskN\_TFSBSig \mid true \bullet \mathrm{pre}\ CreateTaskN\_T$$

It is interesting to understand the proof of this theorem. First of all, as mentioned above, the Z/Eves prover is used to verify our specification. All the proof scripts shown in this thesis are used to help Z/Eves to finish the proof work. Generally, there are two ways to finish a proof [69, Chap. 5]: (a) exploratory proof — directly prove the theorem without any previous plan and address any proof goals returned by the prover; (b) planned proof — carry out a detailed plan for the proof, which is enough to finish the proof by hand, then transfer the plan to a proof script for the prover. To maximise the benefit of proof automation, we adopt the exploratory proof approach in many cases. The general idea for this approach is:

1. Expand terms such as schema references and let Z/Eves prove the proof goal automatically.
2. When Z/Eves is stuck, stop at the proof goals, guide Z/Eves by using or applying related theorems or lemmas to rewrite the proof goals, provide more conditions, etc.
3. Let Z/Eves progress based on the new goal.
4. Repeat step 2 and step 3 until the proof is finished.

For efficiency, it is necessary to expand as few terms as possible in step 1. This can significantly reduce the proof time, especially when the system is complex. This is also one of the reasons for defining our system in parts.

Specifically, we first use the following proof command to expand all necessary terms and then let the prover automatically apply rules and theorems, which are included by Z/Eves, to prove the goal.

$$with\ disabled\ (ContextData)\ prove\ by\ reduce;$$

Meanwhile, because the $ContextData$ is unchanged in this schema, we keep it unexpanded. The prefix $with\ disabled\ (ContextData)$ can achieve this by making the prover ignore $ContextData$, when expanding the terms. Note that as some theorems are rarely used when proving and other theorems are time consuming, Z/Eves disables them by default. This is helpful for improving the efficiency of the proof process; however, it is also one of the reasons why Z/Eves may become stuck

in some cases. As a result, the original proof goal is transferred to the following five goals[2].

1. The tasks known by the system are finite.

$$TASK \setminus ((state \oplus \{(target?, ready)\})^{\sim}(\!| \{nonexistent\} |\!)) \in \mathbb{F}\ TASK$$

   As defined in $CreateTaskN\_T$, the expression $state \oplus \{(target?, ready)\}$ is equal to the post $state$. The left side of the expression indicates all the tasks known by the system after the operation. It should be a finite set as defined in $TaskData$.

2. The $running\_task$ remains the same before and after the operation:

$$(state \oplus \{(target?, ready)\})^{\sim}(\!| \{running\} |\!) = state^{\sim}(\!| \{running\} |\!)$$

   Similarly, the image of $running$ under the inverse function $(state \oplus \{(target?, ready)\})^{\sim}$ represents the $running\_task$ after the operation.

3. The $target?$ task is added into the system by the operation:

$$\begin{aligned} &TASK \setminus ((state \oplus \{(target?, ready)\})^{\sim}(\!| \{nonexistent\} |\!)) \\ &\quad = \{target?\} \cup (TASK \setminus (state^{\sim}(\!| \{nonexistent\} |\!))) \end{aligned}$$

   After the operation, the tasks known by the system should be the same as the known tasks of the pre state of the system plus the created task, which is $target?$.

4. The priority of the $target?$ task is less than or equal to the running task:

$$\begin{aligned} &((state \oplus \{(target?, ready)\})(pt) = ready \land (pt = target? \lor pt \in TASK) \\ &\quad \Rightarrow \\ &\quad priority(running\_task) \geq (priority \oplus \{(target?, newpri?)\})(pt)) \end{aligned}$$

   Comparable to $state$, the post state of the $priority$ function can also be written as: $priority \oplus \{(target?, newpri?)\}$. In this case, this expression is easy to understand.

[2]Because the proof goals are too long to present in this thesis, we only list the most important part here. Please download the Z/Eves project file from the supplementary material and open it with Z/Eves to find the full details.

5. Every state transition made by any task respects the $transition$ relation.

$$(st \in TASK \land \neg\, (state \oplus \{(target?, ready)\})(st) = state(st)$$
$$\Rightarrow$$
$$(state(st), (state \oplus \{(target?, ready)\})(st)) \in transition)$$

It is easy to find that the key to proving both goals 1 & 3 is goal 3. As defined, $TASK \setminus (state^{\sim}(\!|\ \{nonexistent\}\ |\!)$ which is $tasks$, is a finite set. If we can prove goal 3, we can easily show that the union of two finite sets is a finite set. Furthermore, for goal 3, the prover is actually confused by the complex set calculation on the left side of the equation. As these two goals are derived from the constraint of the state schema, we expect they will repeat frequently in the precondition proofs of other schemas. Therefore, we introduce a lemma, $setminUpdate$, to help Z/Eves to discharge this kind of goal automatically. In Z/Eves, it is possible to use the keyword, $rule$, to define an external lemma to help the proof. Z/Eves will use them automatically when the $prove$ command is called.

Theorem 9 (rule setminUpdate)

$$\forall f : TASK \to STATE;\ g : TASK \pfun STATE \bullet$$
$$TASK \setminus ((f \oplus g)^{\sim}(\!|\ \{nonexistent\}\ |\!)) =$$
$$TASK \setminus (f^{\sim}(\!|\ \{nonexistent\}\ |\!)) \setminus (g^{\sim}(\!|\ \{nonexistent\}\ |\!)) \cup$$
$$(\operatorname{dom} g \setminus (g^{\sim}(\!|\ \{nonexistent\}\ |\!)))$$

Similarly, the proof goal 2 is also likely to repeat during the verification. We define another lemma, $runningUpdate$, to improve automation of proof.

Theorem 10 (rule runningUpdate)

$$\forall f : TASK \to STATE;\ g : TASK \pfun STATE \mid$$
$$running \notin \operatorname{ran} g \land$$
$$(f^{\sim}(\!|\ \{running\}\ |\!)) \cap \operatorname{dom} g = \emptyset \bullet$$
$$(f \oplus g)^{\sim}(\!|\ \{running\}\ |\!) = f^{\sim}(\!|\ \{running\}\ |\!)$$

To prove these two lemmas, the $extensionality$ rule in Z/Eves toolkit can be used. The detailed proof script can be found in the supplementary material. After adding these two lemmas before theorem 8 and restarting the proof, we can find that the first three proof goals are discharged automatically.

Proof goal 4 is then given by the constraint in $Task$ schema. Tasks other than $target?$ maintain the requirement that the priority of the running task is at least as great as that of all the ready tasks:

$$\forall\, pt : state^{\sim}(\!|\, \{ready\} \,|\!) \bullet priority(running\_task) \geq priority(pt)$$

A copy of this constraint is also in the assumption part of the goal, and to distinguish pt in these two, Z/Eves renames one from pt to $pt\_\_0$. Therefore, to prove that tasks other than $target?$ obey the constraint, we just need to indicate that $pt\_\_0$ and pt are the same. For $target?$, the priority is defined as $newpri?$, which is specified to be no higher than the priority of $running\_task$ as a precondition of this schema. The rule $applyOverride$ is applied to analyse expressions that contain the operator $\oplus$. Finally, the command $with\ normalization\ prove;$ is used to finish the proof[3]. Thus, the theorem $CreateTaskN\_T\_vc\_ref$ can be proved by following script in Z/Eves.

proof $[CreateTaskN\_T\_vc\_ref]$
$with\ disabled\ (ContextData)\ prove\ by\ reduce;$
$instantiate\ pt\_\_0 == pt;$
$with\ enabled\ (applyOverride)\ prove;$
$apply\ applyOverride;$
$with\ normalization\ reduce;$
■

Second Case of Creating Tasks If the priority assigned to the new task is greater than the priority of the running task, then rescheduling is required. This is achieved by calling the $Reschedule$ schema. The current running task will be moved into the ready state; the new priority and initial context is allocated for the new task, which is then scheduled to be the running task. To reuse $Reschedule$, the variables $st?$, $pri?$ and $tasks?$ are declared and assigned appropriately. Note, the default logical context for the new tasks is $bare\_context$. We do not need to set it separately. Therefore, the schema for the second case of the create task operation can be defined as follows:

$CreateTaskS\_T$

[3]The details about the proof command, $with\ normalization\ prove;$, can be found in Appendix B.

$$
\begin{array}{|l}
\Delta Task \\
target? : TASK \\
newpri? : \mathbb{N} \\
\hline
state(target?) = nonexistent \\
newpri? > priority(running\_task) \\
\exists\, st? : STATE;\ tasks? : \mathbb{F}\, TASK;\ pri? : TASK \to \mathbb{N} \\
\quad \mid st? = ready \\
\quad \land\ tasks? = tasks \cup \{target?\} \\
\quad \land\ pri? = priority \oplus \{(target? \mapsto newpri?)\} \bullet Reschedule \\
\hline
\end{array}
$$

Similarly to the previous case, the signature schema and the precondition theorem can be defined.

$$
\begin{array}{|l}
CreateTaskS\_TFSBSig \\
\hline
Task \\
target? : TASK \\
newpri? : \mathbb{N} \\
\hline
state(target?) = nonexistent \\
newpri? > priority(running\_task) \\
\hline
\end{array}
$$

Theorem 11 (CreateTaskS_T_vc_ref)

$$\forall\, CreateTaskS\_TFSBSig \mid true \bullet \text{pre}\ CreateTaskS\_T$$

This indicates that the new task is unknown to the system before the operation and the priority of the new task is higher than the priority of the running task. This is sufficient and necessary for the precondition of schema $CreateTaskS\_T$.

Deleting Tasks The first case for deleting a task is that it is not the running task: the state of this task—provided it is not the $idle$ task—can be $ready$, $blocked$, or $suspended$, because, normally the handle of the $idle$ task, `xIdleTaskHandle`, is private to the system and impossible for the user to obtain. After the operation, the deleted task will become unknown to the system by deleting it from $tasks$, setting its state to $nonexistent$, and setting its logical context to the $bare\_context$.

It is worth mentioning that in the source code of `vTaskDelete` in FreeRTOS, the context of the deleted task is not actually deleted, but instead moved to the `xTasksWaitingTermination` list. It is the `idle` task that actually performs garbage collection to recover the resources allocated by the system. At this level of abstraction, we consider all this as part of the deletion operation, resetting the $log\_context$ of the deleted task to the $bare\_context$. Note, due to space limitations, we only list the parts of our model which contain something of interest; the rest of the specifications, precondition theorems, and proof scripts can be found from the supplementary material.

Secondly, if the task to be deleted is the running task—but not the $idle$ task—then we remove it from the system. This leaves a vacuum to be filled: we need to schedule another process to use the CPU. We will choose the task in a ready state with the highest priority. However, we cannot use $Reschedule$ to achieve this because the logical context of the running task, which is requested by this operation but not supported by $Reschedule$, will be reset. The output variable $topReady!$ is introduced. The universally quantified expression specifies that the $topReady!$ holds the highest priority. It is worth mentioning here that if there are several solutions, then $topReady!$ is chosen nondeterministically. Similarly, the $tasks$, $state$, $phys\_context$ and $log\_context$ are updated.

$$
\begin{array}{|l}
\hline
DeleteTaskS\_T \\
\Delta Task \\
target? : TASK \\
topReady! : TASK \\
\hline
target? \in tasks \setminus \{idle\} \\
state(target?) \in \{running\} \\
state(topReady!) = ready \\
\forall\, t : state^{\sim}(\!| \{ready\} |\!) \bullet priority(topReady!) \geq priority(t) \\
tasks' = tasks \setminus \{target?\} \\
running\_task' = topReady! \\
state' = state \oplus \{(topReady! \mapsto running), (target? \mapsto nonexistent)\} \\
phys\_context' = log\_context(topReady!) \\
log\_context' = log\_context \oplus \{(target? \mapsto bare\_context)\} \\
\Xi PrioData \\
\hline
\end{array}
$$

The signature schema of this can be obtained as follows.

$DeleteTaskS\_TFSBSig$

$Task$

$target? : TASK$

$target? \in tasks \setminus \{idle\}$

$state(target?) \in \{running\}$

Theorem 12 (DeleteTaskS_T_vc_ref)

$$\forall\, DeleteTaskS\_TFSBSig \mid true \bullet \mathrm{pre}\ DeleteTaskS\_T$$

As mentioned above, the "pre " operator calculates the precondition schema for $DeleteTaskS\_T$, which is the result of $\exists\, Task' \bullet DeleteTaskS\_T \setminus (topReady!)$. When the prover automatically discharges this predicate, it attempts to eliminate existentially quantified variables. Because the post state of the system, $Task'$, has been defined in the operation, the one-point rule[4] is applied to handle them. However, the variable $running\_task'$ and $topReady!$ can only eliminate one of them, because the output variable $topReady!$ is assigned the value of $running\_task'$ in this operation. Therefore, the proof goal will become:

$$\begin{array}{l}
\exists\, running_task' : TASK \bullet \\
\quad Task[log_context := log_context \oplus \{(target?, bare_context)\}, \\
\qquad phys_context := log_context(running_task'), \\
\qquad running_task := running_task', \\
\qquad state := state \oplus (\{(target?, nonexistent)\} \\
\qquad\quad \cup \{(running_task', running)\}), \\
\qquad tasks := tasks \setminus \{target?\}] \\
\quad \land\ (\forall\, st : TASK \mid \\
\qquad \neg\ (state \oplus (\{(target?, nonexistent)\} \cup \\
\qquad\quad \{(running_task', running)\}))(st) = state(st) \bullet \\
\qquad (state(st), (state \oplus (\{(target?, nonexistent)\} \cup \\
\qquad\quad \{(running_task', running)\}))(st)) \in transition) \\
\quad \land\ state(running_task') = ready \\
\quad \land\ (\forall\, t__0 : state^{\sim}(\!|\ \{ready\}\ |\!) \bullet \\
\qquad priority(running_task') \geq priority(t__0)
\end{array}$$

[4] One-point rule: $\exists\, x : X \mid p \bullet q \land x = t \Leftrightarrow p[t/x] \land q[t/x] \land t \in X$, provided that x is not free in t.

Meanwhile, as defined in the specification, one of the highest priority ready tasks is nondeterministically assigned to the variable $running\_task'$. In this case, with the existential-elimination rule, if we can find an instance of these tasks that satisfies this predicate, the proof goal can be verified. Therefore, we introduce the following function (f) to discover a member of a set of tasks that has the highest value of its g, which can be replaced by priority, among other tasks in that set. A label "findDelegate" is assigned to this lemma, which can be referred to during later proofs.

$$
\begin{array}{|l}
f : \mathbb{P}\, TASK \pfun TASK \\
\hline
\emptyset \notin \operatorname{dom} f \\
\langle\!\langle \text{ findDelegate } \rangle\!\rangle \\
\forall\, Task;\ a : \mathbb{P}\, TASK;\ g : TASK \pfun \mathbb{Z} \\
\quad \bullet\ a \in \operatorname{dom} f \land f(a) \in a \land a \subseteq \operatorname{dom} g \land (\forall\, t : a \bullet g(f(a)) \geq g(t))
\end{array}
$$

It is then possible to use this function to find the highest priority task in a ready state and use it to instantiate the $running\_task'$, when the function $priority$ is assigned to the variable g. If we let $p, \exists\, x : X \bullet q$ represent the conditions and the goal of proof above, the predicate to be proved can be considered as

$$p \Rightarrow \exists\, x : X \bullet q. \tag{4.1}$$

Further, let t represent $f(state^{\sim}(\!|\ \{ready\}\ |\!))$. When we instantiate the $running\_task'$ with the delegate, with one-point rule, we have $\exists\, x : X \bullet q \land x = t \Leftrightarrow t \in X \land q[t/x]$, which gives $\exists\, x : X \bullet q \Leftrightarrow \exists\, x : X \bullet q \lor (t \in X \land q[t/x])$. Therefore, equation (4.1) transfers into:

$$p \Rightarrow (\exists\, x : X \bullet q) \lor (t \in X \land q[t/x]). \tag{4.2}$$

Reorganising the equation, the relation

$$p \land \neg\, (t \in X \land q[t/x]) \Rightarrow \exists\, x : X \bullet q. \tag{4.3}$$

can be acquired.

Therefore, applying the proof command "$instantiate\ running\_task' == f(state^{\sim}(\!|\ \{ready\}\ |\!))$; ", a negative copy of this proof goal will be added to the condition part,

of which $running\_task'$ will be replaced by $f(state^{\sim}(\!|\ \{ready\}\ |\!))$. Analysing the negative copy of the goal, we find

$$
\begin{aligned}
Task[log\_context &:= log\_context \oplus \{(target?, bare\_context)\},\\
phys\_context &:= log\_context(f(state^{\sim}(\!|\ \{ready\}\ |\!))),\\
... &\quad \Rightarrow\\
&\quad t \in TASK \wedge\\
&\quad state(t) = ready \wedge\\
&\quad \neg\ priority(f(state^{\sim}(\!|\ \{ready\}\ |\!))) \geq priority(t) \quad (4.4)
\end{aligned}
$$

which conflicts with the definition of function f; therefore, it is **not** $true$. However, according to the implication, if we can prove that the condition is $false$, the result of proof is $true$. As proving this condition is $false$ presents difficulties due to the complexity, an auxiliary theorem, $lDeleteTaskS\_T\_Lemma$, is introduced and proved separately. When we use it to prove $DeleteTaskS\_T\_vc\_ref$, the variable $topReady!$ can be substituted by $f(state^{\sim}(\!|\ \{ready\}\ |\!))$.

Theorem 13 (lDeleteTaskS_T_Lemma)

$$
\begin{aligned}
&\forall\ Task;\ topReady!, target? : TASK\\
&\quad |\ target? \in tasks \setminus \{idle\}\\
&\quad \wedge\ state(target?) \in \{running\}\\
&\quad \wedge\ state(topReady!) = ready\\
&\quad \wedge\ (\forall\ rtsk : state^{\sim}(\!|\ \{ready\}\ |\!) \bullet priority(topReady!) \geq priority(rtsk))\\
&\quad \bullet \neg\ (Task[log\_context := log\_context \oplus \{(target?, bare\_context)\},\\
&\qquad\qquad phys\_context := log\_context(topReady!),\\
&\qquad\qquad running\_task := topReady!,\\
&\qquad\qquad state := state \oplus\\
&\qquad\qquad\qquad (\{(target?, nonexistent)\} \cup \{(topReady!, running)\}),\\
&\qquad\qquad tasks := tasks \setminus \{target?\}]\\
&\qquad \wedge\ (st \in TASK\\
&\qquad\qquad \wedge \neg\ (state \oplus (\{(target?, nonexistent)\} \cup\\
&\qquad\qquad\qquad \{(topReady!, running)\}))(st) = state(st)\\
&\qquad\qquad \Rightarrow (state(st), (state \oplus (\{(target?, nonexistent)\} \cup\\
&\qquad\qquad\qquad \{(topReady!, running)\}))(st)) \in transition)\\
&\qquad \Rightarrow t \in TASK\\
&\qquad\qquad \wedge\ state(t) = ready\\
&\qquad\qquad \wedge \neg\ priority(topReady!) \geq priority(t))
\end{aligned}
$$

Generally, the purpose of Theorem 13 is to prove and notify Z/Eves that for all the states that satisfy the definition of $Task$, based on the precondition of schema $DeleteTaskS\_T$, the proof goal (4.4) is $false$. With this information and the following script, Z/Eves can easily prove the result of Theorem 12 is $true$. $DeleteTaskS\_T\_vc\_ref$ can be continued.

proof $[DeleteTaskS\_T\_vc\_ref]$
$use\ findDelegate[a := state \sim (\!|\ \{ready\}\ |\!),\ g\ :=\ priority];$
$with\ disabled\ (Task)\ prove\ by\ reduce;$
$instantiate\ running\_task' == f(state \sim (\!|\ \{ready\}\ |\!));$
$prove;$
$use\ lDeleteTaskS\_T\_Lemma[topReady! := f(state \sim (\!|\ \{ready\}\ |\!))];$
$prove;$
$instantiate\ t\_\_0 == rtsk;$
$prove;$
■

4.2.4 Executing Tasks

In FreeRTOS, there is no API function for this: once the task is scheduled, it will be executed automatically. However, it is helpful for specifications to show the task being executed, especially when executing the specification with an animator (such as ProZ). In detail, when the processor executes a task, it updates registers, flags, memory location, and so on. We model this by updating the physical context of the processor. Here, we are not interested in the new value after the operation, but want to know that it has changed and the new value has some special property. Therefore, we use a nondeterministic definition again for updating $phys\_context$. Because the schema $ExecuteRunningTask\_T$ describes executing the task, if the new value of $phys\_context$ is different from the original, it will be satisfied.

4.2.5 Suspending/Resuming Tasks

Like creating and deleting, suspending and resuming tasks also have two cases. When the system suspends a ready or blocked task, it does not lead to rescheduling. However, if the task to be suspended is the running task, then the system needs to find another task to take the processor. If a resumed task has a higher priority than the running task, it becomes the new running task, otherwise, it goes

to the ready state. As mentioned above, normally the handle of the $idle$ task is not obtainable. Even though the user may extend the behaviour of the $idle$ task by modifying the `vApplicationIdleHook` function, the $idle$ task must never be suspended [18], and consequently can never be resumed. It is possible to suspend an already suspended task: the system keeps everything the same as before. So, the first case concerns suspending a task that is ready or blocked; the only change necessary is to update the task's state. The following script shows the precondition theorem and proof script of the schema of this case.

Theorem 14 (SuspendTaskN_T_vc_ref)

$$\forall\, SuspendTaskN\_TFSBSig \mid true \bullet \mathrm{pre}\ SuspendTaskN\_T$$

proof $[SuspendTaskN\_T\_vc\_ref]$

$prove\ by\ reduce;$
$apply\ extensionality\ to\ predicate\ TASK \setminus (state \sim (\!| \{nonexistent\} |\!)) = TASK \setminus ((state \oplus \{(target?,\ suspended)\}) \sim (\!| \{nonexistent\} |\!));$
$apply\ extensionality\ to\ predicate\ (state \oplus \{(target?,\ suspended)\}) \sim (\!| \{running\} |\!) = state \sim (\!| \{running\} |\!);$
$instantiate\ pt\_\_0 == pt;$
$prove;$
$apply\ applyOverride;$
$with\ normalization\ prove;$
■

Due to the complication of the proof goal, the final proof command "$with\ normalization\ prove;$" requires a significant amount of time to complete. However, if we use the "$cases$, $next$" commands to separate the proof goals into different cases and then apply "$with\ normalization\ prove;$", it becomes much more efficient.

The second case of the suspend operation is when the suspended task is the running task. Clearly, this leads to rescheduling. This operation ensures that the running task is not the idle task. It selects a target that is ready and is one of the ready tasks with the highest priority (there may be many such tasks). The $Reschedule$ schema is used to achieve the necessary rescheduling. Similar to $DeleteTaskS\_T$, a nondeterministically chosen value is assigned

to $running\_task'$. The prover is therefore confused about its value. An additional theorem, $lSuspendTaskS\_T\_Lemma$, is introduced to help the prover with the pre-condition. Finally, it is also possible to suspend a suspended task. According to the reference manual of FreeRTOS [19], nothing changes when a suspended task is suspended. A single call to `vTaskResume` can resume the task that has been suspended several times. For this reason, in schema $SuspendTaskO\_T$, predicate $\Xi Task$ is used to show that the pre- and post-value of all variables within $Task$ schema are unchanged.

Similarly, the first case of resuming a task does not cause rescheduling. The priority of the resumed task must be no higher than the running task. The task is moved to the $ready$ state and everything else is unchanged. In the second case, the resumed task has a higher priority than the running task, and rescheduling is required. Again, the schema $Reschedule$ is used to approach this.

4.2.6 Changing Priority of Tasks

Because the priority of the $idle$ task is permanently 0, if the target task is $idle$, the $newpri?$ should equal 0. Specifically, to change the priority of tasks, there are three different cases that need to be considered. In the first case, there is no scheduling required, and this follows if one of the following conditions hold:

1. The target is the `running` task and the new priority is at least as high as every other ready task.
2. The target is `ready` and the new priority does not have a greater priority than the running task.
3. The target is the idle task and the new priority is 0.
4. The target is blocked.
5. The target is suspended.

Note that we cannot change the priority of $nonexistent$ tasks. Further, as the set $TASK$ is composed of $running$, $ready$, $blocked$, $suspend$, and $nonexistent$, tasks in these states are disjoint. Therefore, the predicate $state(target?) \neq nonexistent$

implies that the $target?$ is in one of the other four states. That means for conditions related to the $blocked$ and $suspended$ states, we do not need other predicates. Finally, the effect of the operation is to only change the priority of the target, but nothing else. Then, we update the function $priority$ by overriding the priority of the $target?$ task with $newpri?$.

In the second case, the target is a ready task whose new priority is higher than that of the running task. The target displaces the running task as the tasks are rescheduled. Similarly, the $Reschedule$ schema is used to achieve this.

In the third case, similar to the second, rescheduling is required. However, the target task, whose priority we wish to change, is the running task. Meanwhile, the new priority is not the highest of the ready tasks. The schema for this would firstly pick up the task with the highest priority in the ready tasks. It updates the value of the priority of the running task. Finally, it reschedules the system with the $Reschedule$ schema. The variable "$topReady!$" here, similar to the schedule case of delete task and suspend task, is used to represent which ready task holds the highest priority among other ready tasks and would be scheduled as the new running task after the operation. Also, the schema $lChangeTaskPriorityD\_T\_Lemma$ (See Page. 183) is introduced to handle the nondeterministically chosen value of "$running\_task'$".

4.3 Queue Model

Queue is the facility provided by FreeRTOS for communication between tasks. Similar to the task model, to define this model, we need to specify some basic states.

4.3.1 Basic Statements

Firstly, we define $QUEUE$ to represent the queues.

$$[QUEUE]$$

The properties of queues in FreeRTOS, can generally be divided into the following three parts:

1. **Queue data.** In this schema, variables are used to record the basic properties of queues. First, set $queue$ is given to distinguish the queues known to the system from others. Second, it is necessary to know the **maximum** size and the actual size of each queue in the system. Therefore, two functions, q_max and q_size, are respectively used for these, and the domain of these two functions should equal $queue$. Furthermore, for each queue in the system, its actual size cannot exceed its maximum size.

$$
\begin{array}{|l}
\textit{QueueData} \\
\hline
queue : \mathbb{P}\, QUEUE \\
q\_max : QUEUE \pfun \mathbb{N}_1 \\
q\_size : QUEUE \pfun \mathbb{N} \\
\hline
\mathrm{dom}\, q\_max = \mathrm{dom}\, q\_size = queue \\
\forall\, q : QUEUE \mid q \in queue \bullet q\_size(q) \leq q\_max(q) \\
\hline
\end{array}
$$

2. **Waiting data.** As the maximum size of a queue is finite, queues can be full when a task attempts to send an item to them. In this case, FreeRTOS allows the task to wait until spaces become available in the queue. Similarly, when a task wants to receive items from a queue which is empty, the task may also wait for some resources to be available in the queue. Therefore, two functions $wait\_snd$ and $wait\_rcv$ are defined, respectively. Due to the definition of these two functions, a task cannot both be waiting to send and to receive items at the same time. A constraint is specified for this property, that the intersection of the domain of these two functions is the empty set.

$$
\begin{array}{|l}
\textit{WaitingData} \\
\hline
wait\_snd : TASK \pfun QUEUE \\
wait\_rcv : TASK \pfun QUEUE \\
\hline
\mathrm{dom}\, wait\_snd \cap \mathrm{dom}\, wait\_rcv = \emptyset \\
\hline
\end{array}
$$

3. **Releasing data.** According to the implementation of FreeRTOS, when a task is released from a waiting event, it should continue the operation which it was performing before the event. For instance, if the task is blocked by sending

an item to a queue, when it is released from $wait\_snd$ it should continue sending the item to the queue. To achieve this, two assistant functions are provided to indicate if a task has just been released from the waiting event and also record the queue it was dealing with.

$$
\begin{array}{|l}
\hline
QReleasingData \\
release\_snd : TASK \pfun QUEUE \\
release\_rcv : TASK \pfun QUEUE \\
\hline
\mathrm{dom}\ release\_snd \cap \mathrm{dom}\ release\_rcv = \emptyset \\
\hline
\end{array}
$$

Gathering these three schemas, we can define the schema $Queue$ for the properties across each sub-state.

$$
\begin{array}{|l}
\hline
Queue \\
QueueData \\
WaitingData \\
QReleasingData \\
\hline
\mathrm{ran}\ wait\_snd \subseteq queue \\
\mathrm{ran}\ wait\_rcv \subseteq queue \\
\mathrm{ran}\ release\_snd \subseteq queue \\
\mathrm{ran}\ release\_rcv \subseteq queue \\
(\mathrm{dom}\ wait\_snd \cup \mathrm{dom}\ wait\_rcv) \\
\quad \cap (\mathrm{dom}\ release\_snd \cup \mathrm{dom}\ release\_rcv) = \emptyset \\
\hline
\end{array}
$$

First, the range of functions described in $WaitingData$ and $QReleasingData$ are $queue$, because a task cannot send or receive items from a queue which is unknown to the system. Additionally, the tasks of functions in $WaitingData$ and $QReleasingData$ are disjoint, because only when a task is removed from the functions in $WaitingData$, it can then be added to the related function in $QReleasingData$.

Finally, the state data for the queue level model can be defined using these definitions.

$$
\begin{array}{|l}
\hline
TaskQueue \\
Task \\
Queue \\
\end{array}
$$

$$\begin{array}{|l} \hline \mathrm{dom}\ wait\_snd \subseteq state^{\sim}(\!|\ \{blocked\}\ |\!) \\ \mathrm{dom}\ wait\_rcv \subseteq state^{\sim}(\!|\ \{blocked\}\ |\!) \\ \hline \end{array}$$

There are two extra constraints for this schema, which indicate that the state of tasks in the domain of $wait\_snd$ and $wait\_rcv$ is `blocked`.

Initialisation We have defined the initialisation for the $Task$ schema in the previous section. Now, we only need to specify the initialisation for $Queue$, then combine them to obtain the initialisation for the queue model (i.e. $TaskQueue$ schema). Initially, no queues exist in the system, therefore, the initial state of the set $queue$ and all other functions in $QueueData$, $WaitingData$ and $QReleasingData$ are the empty set. Finally, the initialisation of $TaskQueue$ can be defined as

$$\begin{array}{|l} Init\_TaskQueue \\ \hline TaskQueue' \\ \hline Init\_Task \\ Init\_Queue \\ \hline \end{array}$$

To prove the reachability of these initial states, initialisation theorems are also introduced. As the definition of the $Queue$ schema and its sub-state are quite simple, these theorems are easily proved using the command $prove\ by\ reduce$. With the definition of theorem $TaskInit$, the initialisation theorem of $TaskQueue$ can be proved by the following script.

Theorem 15 (TaskQueueInit)

$$\exists\ TaskQueue' \bullet Init\_TaskQueue$$

proof [$TaskQueueInit$]
$use\ TaskInit;$
$prove\ by\ reduce;$
■

4.3.2 Extension

Before specifying operation schemas for the queue model, it is necessary to extend the operations for the task model satisfying the state of the queue model. Firstly, as $\Delta Task$ is overridden in order to check the state transition, we need to override $\Delta TaskQueue$ for this purpose as well.

$$\Delta TaskQueue \mathrel{\widehat=} TaskQueue \land TaskQueue' \land \Delta Task$$

Generally, most task related operations do not need to update information about queues. Schema $ExtendTaskXi$ is given to extend the base state of task related schemas from $\Delta Task$ to $\Delta TaskQueue$, to show that the running task is not released from sending or receiving events and to specify that queue related states are unchanged with $\Xi Queue$.

$ExtendTaskXi$

$\Delta TaskQueue$

$running\_task \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv$
$\Xi Queue$

With this schema, a conjunction relation can be used to easily extend task related schemas to the queue model. For example, to extend $CreateTaskN\_T$ to this level, the following script can be used.

$$CreateTaskN\_TQ \mathrel{\widehat=} ExtendTaskXi \land CreateTaskN\_T$$

Similarly, the following scripts can be used to extend the signature schema of $CreateTaskN\_T$ to the queue level.

$ExtTaskFSBSig$

$TaskQueue$

$running\_task \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv$

$$CreateTaskN\_TQFSBSig \mathrel{\widehat=} ExtTaskFSBSig \land CreateTaskN\_TFSBSig$$

To show that the precondition of $CreateTaskN\_TQ$ is sufficient, theorem 16 is introduced.

Theorem 16 (CreateTaskN_TQ_vc_ref)

$$\forall\, CreateTaskN\_TQFSBSig \mid true \bullet \text{pre}\ CreateTaskN\_TQ$$

As two key components of this theorem, $CreateTaskN\_TQFSBSig$ and $CreateTaskN\_TQ$, are extended from task level, theorem 8 is very helpful to simplify the proving. The command "$use\ CreateTaskN\_T\_vc\_ref$" reuses theorem 8.

Exceptionally, there are some schemas that need to update information about queue related variables. If a task is blocked by a waiting event or it has just been released from a waiting event, it is recorded by waiting or releasing functions in $WaitingData$ and $QReleasingData$. Deleting or suspending it requires updating related data. For instance, when expanding the schema to delete the task in the normal case to this level, information about the target task needs to be removed from $wait\_snd$, $wait\_rcv$, $release\_snd$ and $release\_rcv$.

$DeleteTaskN\_TQ$

$DeleteTaskN\_T$
$\Delta TaskQueue$

$running\_task \notin \text{dom}\ release\_snd \cup \text{dom}\ release\_rcv$
$\Xi QueueData$
$wait\_snd' = \{target?\} \lhd wait\_snd$
$wait\_rcv' = \{target?\} \lhd wait\_rcv$
$release\_snd' = \{target?\} \lhd release\_snd$
$release\_rcv' = \{target?\} \lhd release\_rcv$

Similarly to the general case of extension, the task level schema, $DeleteTaskN\_T$, is reused to introduce constraints and maintain task information. For queue information, variables in $QueueData$ are unchanged, and target task related pairs are removed from functions in $WaitingData$ and $QReleasingData$. Domain anti-restriction operator $\lhd$ is used to approach this. It excludes the set on the left-hand side of an operator from the domain of a relation. Specifically, in our case, pairs whose first element is $target?$ are removed from the functions. If $target?$ does not exist in the domain of the functions, nothing happens to that function.

As only one constraint is added to this new schema, which is the same as $ExtTaskFSBSig$, the signature schema and precondition theorem of $DeleteTaskN\_TQ$ can be given with the same strategy as the creating task case.

4.3.3 Creating and Deleting Queues

The first API function related to queue is `xQueueCreate`, it is used to add new queues into the system. Once a queue is no longer needed, it is deleted by `xQueueDelete` to release the resources. The schema $CreateQueue\_TQ$ and $DeleteQueue\_TQ$ are defined for these operations respectively.

Specifically, to create a queue, the user needs to specify its capacity ($size?$) which should be greater than 0. A new queue ($que?$), which is unknown to the system before the operation, can then be added to the set $queue$. Meanwhile, its maximum and real size can be specified by $size?$ and initial size, which is 0, respectively. In addition, other information should remain unchanged before and after the operation.

In contrast, to delete a queue, we need to remove the information related to the queue from $queue$, q_max and q_size. However, we can only delete a queue, which is known to the system when no task is using it. This means no task is waiting for it and no task has just been released from a waiting event related to the queue.

4.3.4 Sending and Receiving Items

Sending an item to a queue can be represented by increasing the current size of the queue. However, the exact behaviour depends on whether the queue is full, whether there is a task waiting to receive an item from the queue, and whether the waiting task has a higher priority than the running task. It can be divided into four cases, described below. In cases where the running task is released from a waiting-to-send event, it has to continue its attempt at sending and the target queue has to be the queue which the running task attempted to send to before the waiting event.

The first is the most general case. There is space left in the queue, which means the running task can successfully send an item to the queue. Meanwhile, there are no other tasks waiting to receive an item from the queue. Therefore, the running task can send the item to the queue normally. Consequentially, after the operation, the size of $que?$ should be increased by 1; the running task should be removed from the function $release\_snd$ and all other data should remain unchanged.

$$
\begin{array}{|l}
\hline
QueueSendN\_TQ \\
\Delta TaskQueue \\
que? : QUEUE \\
topReady! : TASK \\
\hline
running\_task \notin \operatorname{dom} release\_rcv \\
running\_task \in \operatorname{dom} release\_snd \\
\quad \Rightarrow que? = release\_snd(running\_task) \\
que? \in queue \\
q\_size(que?) < q\_max(que?) \\
que? \notin \operatorname{ran} wait\_rcv \\
\Xi Task \\
queue' = queue \\
q\_max' = q\_max \\
q\_size' = q\_size \oplus \{(que? \mapsto q\_size(que?) + 1)\} \\
\Xi WaitingData \\
release\_snd' = \{running\_task\} \lhd release\_snd \\
release\_rcv' = release\_rcv \\
topReady! = running\_task \\
\hline
\end{array}
$$

The schema $QueueSendW\_TQ$ is introduced for the second case. The postfix W used here represents that there are waiting tasks. In detail, there is also space in the queue. However, there are tasks waiting to receive an item from the queue and the priority of the highest priority task ($topReady!$) is lower than or equal to the current running task. As a result, $topReady!$ will be woken up and recorded in $release\_rcv$, as it has just been released from a waiting-to-receive event. In other respects, it is the same as the normal case. As the $topReady!$ is chosen nondeterministically from the domain of $wait\_rcv$, the assistant theorem $lQueueSendW\_TQ\_Lemma$ is used to help with the proof.

Third, similar to the previous case, there is space in the queue and tasks wait-

ing to receive, but the priority of the highest priority waiting task ($topReady!$) is greater then the running task. The schema $QueueSendWS\_TQ$ represents this case. When the waiting task is woken up, it is rescheduled as the new running task. Therefore, the schema for this case uses the schema $Reschedule$ to maintain task related information and a similar script to the previous case is used to manage queue related data.

In the fourth case, when the target queue is full (the schema $QueueSendF\_TQ$ is defined for this case), the running task is blocked. Similar to the suspending running task, this leads to rescheduling, which is achieved by using the schema $Reschedule$. Furthermore, because the running task is blocked by a waiting-to-send event, the maplet $(running\_task \mapsto que?)$ is added to the function $wait\_snd$. Finally, running task related information also needs to be removed from $release\_snd$. Similarly, the new running task is nondeterministically selected. The theorem $lQueueSendF\_TQ\_Lemma$ is introduced to help with the proof.

Just like sending an item to a queue, receiving an item from a queue can be specified as decreasing the size of the queue and there are four cases for the receiving operation: (a) Normal case ($QueueReceiveN\_TQ$). There are items available in the queue and no other tasks are waiting to send an item to the queue. The size of the queue is decreased by one; we remove running task related information from $release\_rcv$ and keep everything else unchanged; (b) Waiting case ($QueueReceiveW\_TQ$). There are items available in the queue and some tasks waiting to send an item to the queue. Moreover, the priority of the highest priority waiting task is no higher than the running task. The highest priority waiting task is moved to the $ready$ state and recorded in $release\_snd$. The schema maintains other variables as in the first case; (c) Waiting and scheduling case ($QueueReceiveWS\_TQ$). There are items in the queue and some tasks waiting to be sent. The highest priority waiting task has a higher priority than the running task and rescheduling is required. The schema $Reschedule$ is used for this. Other variables are maintained in a similar way to the previous case; (d) Queue is empty case ($QueueReceiveE$). The running task is blocked by the

waiting-to-receive event. The running task is rescheduled to the $blocked$ state by the $Reschedule$ schema and recorded in the function $wait\_rcv$. Further, the operation removes running task related information from $release\_rcv$ and keeps other variables unchanged.

4.4 Time Model

At this level of modelling, we take time properties into consideration. As a real-time multi-tasking system, this is crucial. For instance, the processor is shared by tasks with the same priority based on time sharing. Different tasks may also need to cooperate with each other based on the same system clock, etc. As in previous models, to illustrate the time model, we start from the basic information.

4.4.1 Basic Statements

A constant, $slice\_delay$ is defined to represent the unit of time for each time slice, which is specified as 1 in this case. A label, "disabled slice_delay_def", is given to this lemma. With the `disabled` mark, this lemma is automatically omitted by Z/Eves during proofs.

$$slice\_delay : \mathbb{N}$$

$$\langle\!\langle \text{ disabled slice\_delay\_def } \rangle\!\rangle$$
$$slice\_delay = 1$$

Compared with the task and queue models, the base state of the time model is very simple, as it only contains four variables. First, the system clock is represented by a variable, $clock$. Second, when tasks block themselves by calling delay API functions, the set $delayed\_task$ is used to mark them. The key difference between the `blocked` and the `suspended` state is blocking time. Tasks in the `suspended` state can only be resumed by certain types of events. However, tasks in a `blocked` state can be woken up by both events and time. Third, to record how long a `blocked` task will be blocked, a function $wait\_time$ is introduced. In a single-core multi-task operating system, CPU time is divided into individual time slices; tasks with the same priority can then take one slice in turn, to share CPU time equally. In FreeRTOS, a

similar strategy is used for CPU sharing. For this purpose, $time\_slice$ is defined to show the number of time slices that have passed.

$$
\begin{array}{|l}
\textit{Time} \\
\hline
clock : \mathbb{N} \\
delayed\_task : \mathbb{P}\ TASK \\
wait\_time : TASK \pfun \mathbb{N} \\
time\_slice : \mathbb{N} \\
\hline
\forall\, t : \operatorname{dom} wait\_time \bullet wait\_time(t) \geq clock \\
\hline
\end{array}
$$

In $wait\_time$, each task is mapped to the time at which it should finish waiting, so each element of the range should be greater than (after) the system clock, which is $clock$.

Because the range of $wait\_time$ indicates the time tasks need to wait, no members should be less than the system clock, which is $clock$.

Similar to the schema $TaskQueue$, the state schema for this level of model can be specified by combining $TaskQueue$ and $Time$.

$$
\begin{array}{|l}
\textit{TaskQueueTime} \\
\hline
TaskQueue \\
Time \\
\hline
\langle delayed\_task, \operatorname{dom} wait\_snd, \operatorname{dom} wait\_rcv \rangle \text{ partition } \operatorname{dom} wait\_time \\
delayed\_task \subseteq state^{\sim}(\!|\ \{blocked\}\ |\!) \\
\hline
\end{array}
$$

As the function $wait\_time$ records blocking time for each blocked task, its domain has to contain all the tasks in `blocked` state (i.e., $delay\_task$, $\operatorname{dom} wait\_snd$ and $\operatorname{dom} wait\_rcv$). Similar to the relation between $\operatorname{dom} wait\_snd$ and $\operatorname{dom} wait\_rcv$ described in Sect. 4.3.1, once a task is blocked by an event, it cannot continue its execution and be blocked by another event. The intersection of each two of these sets should be the empty set. Therefore, we can define that $delay\_task$, $\operatorname{dom} wait\_snd$ and $\operatorname{dom} wait\_rcv$ partition $\operatorname{dom} wait\_time$. As mentioned above, the set $delayed\_task$ marks tasks which are blocked by themselves. All tasks in $delayed\_task$ are in the `blocked` state.

Additionally, it is clear that the state of tasks in the domain of $wait\_time$ have to be $blocked$. The prover, however, cannot get this assertion directly. Theorem 17 is provided to guide the prover. The proof is also quite simple; we need to indicate that: (a) The tasks in the domain of $wait\_time$ are also tasks in $delay\_task$, $\mathrm{dom}\, wait\_rcv$ or $\mathrm{dom}\, wait\_snd$; (b) The state of tasks in $delay\_task$, $\mathrm{dom}\, wait\_rcv$ and $\mathrm{dom}\, wait\_snd$ is $blocked$. After this, Z/Eves can handle the rest of the work using the proof command "$with\ normalization\ rewrite$". Specifically, the first three proof commands in the following script are used to expand $TaskQueueTime$ and $TaskQueue$ schemas, as they contain the necessary information, for instance, the relation between the domain of $wait\_time$ and the other three sets ($delay\_task$, $\mathrm{dom}\, wait\_rcv$ and $\mathrm{dom}\, wait\_snd$), and the state of the tasks in these sets. The fourth and fifth commands indicate to the prover that the union of $delay\_task$, $\mathrm{dom}\, wait\_rcv$ and $\mathrm{dom}\, wait\_snd$ equals the domain of $wait\_time$ and t can be a member of one of these three sets. The final five proof commands request that the prover rewrites the proof goal with the theorem $inPower$[5] and then proves that the state of the task t is $blocked$.

Theorem 17 (rule domTime)

$$\forall\ TaskQueueTime;\ t : TASK \mid t \in \mathrm{dom}\, wait\_time \bullet t \in state^{\sim}(\!|\ \{blocked\}\ |\!)$$

proof $[domTime]$
$invoke\ TaskQueueTime;$
$invoke\ TaskQueue;$
$prove;$
$apply\ extensionality\ to\ predicate\ delayed\_task$
$\cup\,(\mathrm{dom}\ wait\_rcv \cup \mathrm{dom}\ wait\_snd) = \mathrm{dom}\ wait\_time;$
$instantiate\ y == t;$
$with\ enabled\ (inPower)\ prove;$
$instantiate\ e == t;$
$instantiate\ e\_\_0 == t;$
$instantiate\ e\_\_1 == t;$
$with\ normalization\ rewrite;$
■

[5] $X \in \mathbb{P}\, Y \Leftrightarrow (\forall\, e : X \bullet e \in Y)$

Initialisation Initially, the system clock is 0 and in the first CPU time slice. Moreover, there is no task blocked, so the set $delayed\_task$ and the function $wait\_time$ should be empty.

$$
\begin{array}{|l}
\hline
Init\_Time \\
Time' \\
\hline
clock' = 0 \\
delayed\_task' = \emptyset \\
wait\_time' = \emptyset \\
time\_slice' = slice\_delay \\
\hline
\end{array}
$$

Combining the schema $Init\_Time$ and $Init\_TaskQueue$, the initialisation schema for the time model can be generated. With the help of theorem 15, the initialisation theorem for this model is also easy to prove.

4.4.2 Extension

Similar to the queue model, we need to override the schema $\Delta TaskQueueTime$ to enable the state transition check for related schemas and define an extension schema to help upgrade the schemas for the queue model to this level, which expands the base state to $\Delta TaskQueueTime$ and keeps variables in $Time$ unchanged. Most schemas can be easily upgraded by a conjunction between the queue level schema and the extension schema, like the extension for the general case described in Sect. 4.3.2. However, there are schemas that need to update variables in the $Time$ schema as well.

- Similar to the queue model, deleting and suspending operations need to remove the target task from $delayed\_task$ and $wait\_time$, if it is blocked;
- Schemas that unblock a task from the `blocked` state also need to remove the unblocked task from the function $wait\_time$;
- Finally, schemas that block the running task need to add blocking time information into the function $wait\_time$.

For instance, when a queue is full, sending items to that queue blocks the running task. Originally, the API function `xQueueSend` blocked the running task for a

period of time. To simplify our model, the running task will block until a specified time ($wtime?$ in the specification). Therefore, an extra precondition is given, that the waking time is later than the current clock. The maplet, $(running\_task \mapsto wtime?)$, is appended to $wait\_time$ to record this. (Signature schema and the precondition theorem proof of this schema can be found on Page. 212 and in the supplementary material as well.)

$$
\begin{array}{|l}
\hline
QueueSendF\_TQT \\
\Delta TaskQueueTime \\
QueueSendF\_TQ \\
wtime? : \mathbb{N} \\
\hline
wtime? > clock \\
clock' = clock \\
delayed\_task' = delayed\_task \\
wait\_time' = wait\_time \oplus \{(running\_task \mapsto wtime?)\} \\
time\_slice' = time\_slice \\
\hline
\end{array}
$$

4.4.3 Delaying Tasks

In FreeRTOS, there are two API functions for delaying tasks, `vTaskDelay` and `vTaskDelayUntil`. One delays tasks for a certain period and the other delays tasks until a specific time. To simplify the model, we only modelled one of them, specifying `vTaskDelayUntil`.

To delay a task, firstly, delaying time ($wtime?$) needs to be specified as later than the system clock. As the running task is blocked, rescheduling is requested. Like other schemas requesting rescheduling, the schema $Reschedule$ is used to do this. Furthermore, $delayed\_task$ and $wait\_time$ have to be updated to record this information as well.

4.4.4 Checking Delayed Tasks

Once the blocking time of a blocked task has expired, it needs to be woken automatically by the system. This is performed by the function `prvCheckDelayedTasks` in FreeRTOS. When the scheduler increases the clock ticks, this function is called to check if there are any tasks that need to be woken up. If there are, these tasks

are moved to the ready list for scheduling. Two cases are used to model this. When the priority of a woken task is no higher than the running task, it is moved from `blocked` state to `ready` state. However, when its priority is higher than the running task, rescheduling is requested.

Specifically, in our model, when tasks' blocking time expires, we unblock them in the order of priority. Therefore, the next unblocked task ($topWaiting!$) should have the earliest waking time. If there is more than one task that holds the same wake up time, it should be the highest priority task first. Depending on whether its priority is greater than the running task or not, there are two separate schemas. For normal cases, it does not need to reschedule. For task related variables, we only need to override the $state$ of $topWaiting!$ to $ready$ and keep everything else unchanged. Meanwhile, $topWaiting!$ has to be removed from one of the block related lists (i.e., $delay\_task$, $wait\_snd$, $wait\_rcv$) and $wait\_time$. In addition, we also need to update the system clock to the wake up time of $topWaiting!$ to show that time has passed. This is because at this level of abstraction, we are not interested in the behaviour of the system at each time tick. Our main objective is verifying the correctness of the API functions. In this way, we can ensure that tasks are unblocked in the correct order, while focusing on the behaviour of the operation and ignoring the trivial details (at this level of abstraction) of the clock ticks.

$$
\begin{array}{|l}
\hline
CheckDelayedTaskN\_TQT \\
\Delta TaskQueueTime \\
topWaiting! : TASK \\
\hline
running\_task \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv \\
topWaiting! \in \operatorname{dom} wait\_time \\
\forall\, wt : \operatorname{dom} wait\_time \bullet wait\_time(topWaiting!) \leq wait\_time(wt) \\
\forall\, wt : \operatorname{dom} wait\_time \mid wait\_time(wt) = wait\_time(topWaiting!) \\
\quad \bullet\ priority(topWaiting!) \geq priority(wt) \\
priority(topWaiting!) \leq priority(running\_task) \\
\Xi TaskData \\
state' = state \oplus \{(topWaiting! \mapsto ready)\} \\
\Xi ContextData \\
\Xi PrioData \\
\Xi QueueData \\
wait\_snd' = \{topWaiting!\} \lhd wait\_snd \\
wait\_rcv' = \{topWaiting!\} \lhd wait\_rcv \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\Xi QReleasingData \\
clock' = wait\_time(topWaiting!) \\
delayed\_task' = delayed\_task \setminus \{topWaiting!\} \\
wait\_time' = \{topWaiting!\} \lhd wait\_time \\
time\_slice' = time\_slice
\end{array}
$$

In contrast, if the priority of $topWaiting!$ is higher than the running task, rescheduling is required. The schema $Reschedule$ is used to maintain task related variables. For other variables, it is the same as the normal case.

As $topWaiting!$ is selected nondeterministically, assistant lemmas, $lCheckDelayedTaskN\_TQT\_Lemma$ and $lCheckDelayedTaskS\_TQT\_Lemma$, are introduced to help prove the precondition theorems of these two cases respectively.

4.4.5 Time-Sharing

As a multi-task operating system, if there are any ready tasks that have the same priority as the running task, they will share the CPU time. In FreeRTOS, this is implemented by an interrupt service routine. When a time slice passes, it will trigger an internal interrupt to check whether it is necessary to perform rescheduling. If there are other tasks holding the same priority as the running task, the next task in the ready list for that priority will be rescheduled. In our specification, the schema $TimeSlicing\_TQT$ is specified for this case. It nondeterministically selects the next running task from ready tasks, which have the same priority as the current running task. The schema $Reschedule$ can be used for rescheduling. Then $time\_slice$ is increased to indicate that one time slice has passed. However, if the running task is the only task that has the highest priority, it can continue to occupy CPU time. In this case, the schema $NoSlicing\_TQT$ only needs to increase the $time\_slice$ and keeps everything else unchanged.

4.5 Mutex Model

As described in Sect. 1.2.2, semaphores and mutexes are used to manage shared resources and are special queues. There are two types of semaphores

in FreeRTOS, counting semaphores and binary semaphores. The counting semaphores allow more than one task to hold the semaphore, which is specified by its size. However, the binary semaphores are similar to mutexes, which only allow one task to hold the semaphore. The difference between binary semaphores and mutexes is that binary semaphores do not support priority inheritance when competition happens. These two types of semaphores have similar properties. To simplify the specification, we only model the binary one.

4.5.1 Basic Statement

Basic information about semaphores and mutexes is gathered in the schema $MutexData$, which defines that semaphores and mutexes are members of $QUEUE$. As a task holds a mutex, it can take the mutex repeatedly. The function $mutex\_holder$ is introduced to record the mutex holder for each mutex. Meanwhile, it is also necessary to know how many times the mutex holder retakes the mutex; because, it has to return the mutex the same number of times to actually return the mutex. There are three constraints for $MutexData$: (a) The intersection of $semaphore$ and $mutex$ should be the empty set. Although, in FreeRTOS, the mutex is treated as a special semaphore, they have different properties and operations. Therefore, we separate them into two disjoint sets. (b) The domain of the function $mutex\_recursive$ has to be $mutex$, as it records how many time a mutex holder repeatedly takes the mutex. (c) If a mutex is not held by a task, its $mutex\_recursive$ has to be 0.

$$
\begin{array}{|l}
\hline
MutexData \\
\hline
semaphore : \mathbb{P}\, QUEUE \\
mutex : \mathbb{P}\, QUEUE \\
mutex\_holder : QUEUE \pfun TASK \\
mutex\_recursive : QUEUE \pfun \mathbb{N} \\
\hline
mutex \cap semaphore = \emptyset \\
\dom\, mutex\_recursive = mutex \\
\forall\, m : mutex \bullet m \notin \dom\, mutex\_holder \Leftrightarrow mutex\_recursive(m) = 0 \\
\hline
\end{array}
$$

To enable the priority inheritance mechanism, the schema $OriginalPrioData$ is provided, which has only one function $base\_priority$ recording the original priority of a mutex holder.

$$
\begin{array}{|l}
\hline
\textit{OriginalPrioData} \\
\textit{base\_priority} : \textit{TASK} \pfun \mathbb{N} \\
\hline
\end{array}
$$

Similar to queues, when a task is released from waiting to take a mutex event, it has to continue its attempt. However, because only the mutex holder can give the mutex back, it is impossible to block a task from giving back a mutex. The schema $MReleasingData$ is defined to record the task which has just been released from waiting to take a mutex.

$$
\begin{array}{|l}
\hline
\textit{MReleasingData} \\
\textit{release\_mutex} : \textit{TASK} \pfun \textit{QUEUE} \\
\hline
\end{array}
$$

Based on these definitions, the constraints between them can be defined in the schema $Mutex$.

$$
\begin{array}{|l}
\hline
\textit{Mutex} \\
\textit{MutexData} \\
\textit{OriginalPrioData} \\
\textit{MReleasingData} \\
\hline
\dom \textit{base\_priority} = \ran \textit{mutex\_holder} \\
\ran \textit{release\_mutex} \subseteq \textit{mutex} \\
\hline
\end{array}
$$

Finally, the base state of the mutex model can be defined as follows. First, semaphores and mutexes are special cases of queues, which have a maximum size of 1. Second, if a mutex is held by a task, its size should be 0. Therefore, the domain of $mutex\_holder$ is all the mutexes with size 0. Third, only tasks known by the system can take and hold a mutex, which means the range of $mutex\_holder$ is a subset of $tasks$. Priority inheritance is used to avoid a higher priority task being blocked by a lower priority mutex holder. If a mutex holder inherits a priority from another task, the new priority has to be greater than its original priority. Furthermore, only the holder of a mutex or a semaphore can return the mutex or the semaphore, so it is impossible for a task to be blocked and released by a sending event. Finally, as mutexes are a subset of queues and taking a mutex is implemented by receiving from a queue, the function $release\_mutex$ should be a subset of the function $release\_rcv$ as well.

$$
\begin{array}{|l}
\hline
TaskQueueTimeMutex \\
TaskQueueTime \\
Mutex \\
\hline
semaphore \subseteq queue \\
\forall\, s : semaphore \bullet q\_max(s) = 1 \\
mutex \subseteq queue \\
\forall\, m : mutex \bullet q\_max(m) = 1 \\
\mathrm{dom}\, mutex\_holder = \{m : mutex \mid q\_size(m) = 0\} \\
\mathrm{ran}\, mutex\_holder \subseteq tasks \\
\forall\, mh : \mathrm{ran}\, mutex\_holder \bullet priority(mh) \geq base\_priority(mh) \\
\forall\, ms : mutex \cup semaphore \bullet ms \notin \mathrm{ran}\, wait\_snd \cup \mathrm{ran}\, release\_snd \\
release\_mutex \subseteq release\_rcv \\
\hline
\end{array}
$$

Initially, no semaphore or mutex exists in the system. All variables in $Mutex$ should be empty at this point.

4.5.2 Extension

A similar strategy of extension to that in Sect. 4.4.2 can be used to upgrade the specification of the time model to the mutex model. The schema $ExtendTQTXi$ is introduced, which expands the base state and leaves variables in $Mutex$ unchanged. However, there are some operations that need to update variables in $Mutex$.

- A mutex holder cannot be deleted by the system. The resource locked by the mutex would no longer be available for other tasks, if the mutex holder is deleted before it releases the mutex.
- Semaphores and mutexes are special queues, which means the queue operations may treat a semaphore or mutex as a normal queue. However, the operations for queues are not actually correct for semaphores and mutexes. For instance, when creating a queue, it should be empty by default. Then the user can use it for communication. However, for a semaphore or a mutex, the user wants it to be full initially as this indicates that a resource is available. It is important to prevent queue operations from dealing with the semaphores and mutexes. Therefore, the constraint "$que? \notin semaphore \cup mutex$" is added to all schemas for queue operations.

- Similarly, as the function $release\_mutex$ is a sub-function of $release\_rcv$, it is also vital to indicate that the running task is not released from taking a mutex for the queue receiving schemas.
- Finally, the changing priority operations are the most complex extensions for this level. In the previous model, we have three cases for changing task priorities. At this level, it depends on whether: (a) the target task is holding a mutex, or not; (b) the target task is inheriting a priority, or not; (c) the new priority is higher than the inherited priority and the priority of the running task, or not. Each case can be subdivided into three more cases. Specifically, for tasks not holding a mutex, we keep $Mutex$ unchanged and append the precondition to indicate the target task is not a member of the domain of $base\_priority$, which equals the domain of $mutex\_holder$, for the three cases, $NNotHolder$ [6], $SNotHolder$ and $DNotHolder$. For tasks holding a mutex but not inheriting a priority, the schemas for the three cases ($NNotInherited$, $SNotInherited$ and $DNotInherited$) have to update the information of $base\_priority$ as well. When the target task is a member of the domain of the base priority and is equal to its current priority, it implies that the task is not inheriting a priority. The preconditions are appended to extend the three cases of changing task priority. Finally, in the case of the target inheriting a priority from another task, (a) if the new priority is not greater than its inheriting priority, the operation updates the base priority of the target task only; (b) meanwhile, if its new priority is greater than its inheriting priority but it is lower than or equal to the running task, it is based on the normal case of changing the priority of a task and updates the base priority of the target task; (c) however, if its new priority is greater than that of the running task, its base priority is updated and rescheduling is requested; the schema $ChangeTaskPriorityS\_TQT$ is used to simplify the specification. The schema $InheritedN$, $InheritedU$ and $InheritedS$ are defined for this case.

[6] Due to the length of these schema names, we present the postfix for each schema here only; i.e. the postfix $NNotHolder$ represents the schema $ChangeTaskPriorityNNotHolder\_TQTM$

4.5.3 Creating and Deleting Semaphores and Mutexes

As described in Sect. 4.5.1, initially there is no semaphore or mutex in the system. FreeRTOS provides the API functions `vSemaphoreCreateBinary` and `xSemaphoreCreateMutex` to introduce a new binary semaphore and mutex to the system. Later, when these structures are no longer needed, they can be removed from the system by the API function `vSemaphoreDelete`.

Semaphores and mutexes are special queues, which have a maximum capacity of 1. The behaviour of their creating and deleting operations is similar to that of queues. In FreeRTOS, creating a semaphore or mutex actually creates a new queue with capacity 1. It then sends an item to fill the semaphore or mutex, which makes the semaphore or mutex available. Meanwhile, the API function `vSemaphoreDelete` is directly defined by `vQueueDelete`.

We also try to follow this strategy to reuse existing specifications in new schemas. This not only simplifies the definition of new schema, but also dramatically reduces the complexity of the verification, because the precondition theorem for existing schemas can be used directly in the proof of the precondition theorem for the new schema. Therefore, when defining the schemas $DeleteBinarySemaphore\_TQTM$ and $DeleteMutex\_TQTM$, the schema $DeleteQueue\_TQT$ is used to manage system information before the mutex model. The deleted semaphore and mutex are then removed from related variables, $semaphore$, $mutex$ and $mutex\_recursive$ respectively. It is worth noting that when deleting a mutex from the system, it should not be held by any task.

However, when creating operations, as the initial size of the new semaphore and mutex is full instead of empty, $CreateQueue\_TQT$ cannot be used. The schema $CreateBinarySemaphore\_TQTM$ and $CreateMutex\_TQTM$ add a new queue to the $queue$ and set its capacity and size to 1. The queue is also added to $semaphore$, $mutex$ and $mutex\_recursive$. All other variables remain unchanged.

4.5.4 Taking Mutexes

As a special queue, taking a mutex can be treated as receiving an item from the mutex. Basically, we attempt to reuse specifications for the queue receiving operation in our specification for mutex taking. In Sect. 4.3.4, four cases are defined for receiving an item from a queue. However, the second and third cases are impossible for mutex taking, as they are introduced for cases where there are tasks waiting to send an item to the queue and there is no wait to send events for semaphores and mutexes. Based on the normal and empty cases of the receiving operation, the specification for the mutex taking operation can be defined. Specifically, there are two sub-cases for the normal case, depending on whether the running task previously held a mutex.

- $MutexTakeNnonInh\_TQTM$ If the running task did not previously hold a mutex, which means its base priority was not initialised, we need to initialise it with the value of its current priority.
- $MutexTakeNInh\_TQTM$ Otherwise, the base priority of the running task should not be changed.

In addition, the remaining part of these two sub-cases are the same: (a) They reuse the schema $QueueReceiveN\_TQT$ to receive the item from the mutex, which shows that the running task takes the mutex; (b) Set the running task as the holder of the mutex; (c) Increase the value of $mutex\_recursive$; (d) Finally, remove information about the running task from $release\_mutex$ to enable the running task to execute other operations.

When the mutex is not available, the running task will be blocked by waiting for a receiving event. Depending on the relationship between the priorities of the running task and the mutex holder, there are also three cases.

- $MutexTakeEnonInh\_TQTM$ If the priority of the running task is not higher than that of the mutex holder, the mutex holder keeps its priority and the running task is replaced by a ready task with the highest priority, which is

the same as the empty case of the queue receiving operation. The schema $QueueReceiveE\_TQT$ is used to achieve this and running task related information is also removed from $release\_mutex$.

- On the other hand, if the priority of the running task is higher than the mutex holder, the mutex holder inherits the priority of the running task and the running task is blocked.
  - $MutexTakeEInheritReady\_TQTM$ If the state of the mutex holder is `ready`, it becomes the new running task.
  - $MutexTakeEInheritHolder\_TQTM$ Otherwise, a task with highest priority in the `ready` state is selected as the new running task.

  In these cases, the schema $QueueReceiveE\_TQT$ cannot be used, because the priority of the mutex holder is updated. However, $Reschedule$ is used to manage rescheduling. Similarly, it also needs to remove information about the running task from related release functions (i.e., $release\_rcv$ and $release\_mutex$). As the running task is blocked, data about the blocking time needs to be appended to the $wait\_time$ function as well. Other variables remain unchanged.

Finally, the last case, $MutexTakeRecursive\_TQTM$, is for recursively taking the mutex, which is the simplest case. It increases the value of the mutex in $mutex\_recursive$ and keeps everything else the same.

In summary, taking mutexes has six cases in total.

4.5.5 Giving Mutexes

Similar to taking mutexes, when a mutex holder gives the mutex back, it actually sends an item back to the mutex, even if the item size is zero. Therefore, the schemas for queue sending are reused here to develop new specifications for giving mutexes. As we define the domain of $base\_priority$ to be equal to the range of the $mutex\_holder$, when a task gives all its mutexes back, its base priority has to be removed as well. This can happen to every case of giving mutexes. The schema $basePriorityMan$ is introduced to perform base priority management. It checks

whether the running task is giving its last mutex. If it is, $basePriorityMan$ removes the running task related pairs from $base\_priority$; otherwise, $base\_priority$ remains unchanged.

$$
\begin{array}{|l}
\hline
basePriorityMan \\
\Delta TaskQueueTimeMutex \\
mut? : QUEUE \\
\hline
running\_task \in \mathrm{ran}(\{mut?\} \lhd mutex\_holder) \\
\quad \Rightarrow \Xi OriginalPrioData \\
running\_task \notin \mathrm{ran}(\{mut?\} \lhd mutex\_holder) \\
\quad \Rightarrow base\_priority' = \{running\_task\} \lhd base\_priority \\
\hline
\end{array}
$$

The first case of giving mutexes ($MutexGiveNRecursive\_TQTM$) is the simplest one, which handles the recursive return. When the mutex holder has taken the mutex several times, the mutex holder has to return the mutex the same number of times to make the mutex available for other tasks. When the value of the mutex in $mutex\_recursive$ is greater than 1, the schema $MutexGiveNRecursive\_TQTM$ decreases $mutex\_recursive$ by 1 each time. In addition, there are eight cases for giving mutexes. The behaviour depends on: (a) Whether there are tasks waiting to take the mutex; (b) Whether the priority of the mutex holder is inherited from other task; (c) Which priority is the highest (the base priority of the running task; the priority of the running task, the priority of the top priority ready task or the priority of the top priority waiting task). Table 4.1 and 4.2 illustrate the relationship between each case[7].

Table 4.1 shows that based on the normal case of the queue sending operation there are three different cases. Specifically, for the first case ($NnonInh$ case), there are no tasks waiting to take the mutex and the mutex holder does not inherit priority from other tasks. As the mutex holder is the current running task, we know its priority is the highest of the ready tasks. Therefore, there is no rescheduling request. We just need to send an item to the mutex to indicate that the mutex holder returns the mutex. Meanwhile, information about the mu-

[7]Due to the length of schema names, only the postfixes for each schema are presented in the table; i.e., the postfix $NnonInh$ stands for the schema $MutexGiveNnonInh\_TQTM$

Table 4.1: The constraints for giving mutexes (no waiting tasks)

Inh	N	Y	
Highest Prio	-	Base	Ready
Postfix	$NnonInh$	$NInhN$	$NInhS$

Table 4.2: The constraints for giving mutexes (with waiting tasks)

Inh	N		Y		
Highest Prio	Run	Waiting	Base	Ready	Waiting
Postfix	$WnonInhN$	$WnonInhS$	$WInhN$	$WInhSR$	$WInhSW$

tex in $mutex\_holder$ and $mutex\_recursive$ has to be reset. Finally, the schema $basePriorityMan$ is used to manage the base priority of the mutex holder. The schema $MutexGiveNnonInh\_TQTM$ uses $QueueSendN\_TQT$ to send an item to the mutex. If the mutex holder inherits a priority from another task, its priority has to be reset to its original priority when it gives the mutex back. The second and third cases in this group are defined for this. The difference between them is when the original priority of the mutex holder (i.e., the running task) is lower than the priority of a ready task, rescheduling is requested. Unfortunately, the schema $QueueSendN\_TQT$ cannot be used, as the priority of the mutex holder needs to be modified, which is not covered by the definition of $QueueSendN\_TQT$.

Similar to the last group of cases, the cases in Table 4.2 are based on waiting, and waiting and rescheduling cases of queue sending operations. There are tasks waiting to take the mutex. It is necessary to wake these tasks when the mutex holder returns the mutex. When the mutex holder does not inherit a priority, these cases are extensions from $QueueSendW\_TQT$ and $QueueSendWS\_TQT$. Otherwise, the priority of the mutex holder needs to be revised to its original priority. Moreover, depending on the relationship between the base priority of the running task, the priority of the top priority ready task and the priority of the top priority waiting task, each case needs to decide which task should be the new running task after the operation (i.e., the highest priority task can be executed).

4.6 Summary of Interface

The preconditions for the interface, and the API function mapping can be found in Appendix C. When we define the schemas for API functions, in order to simplify the specification, we use different schemas to define the different cases of API functions. We use disjunction to connect them together to get the schema that represents the API function in FreeRTOS. The precondition for these new schemas is also obtained from the preconditions of the old schemas, which are also disjoint. For instance, the FreeRTOS API function for creating a task, `xTaskCreate`, is represented by the schema $CreateTask\_T$, which has two cases, $CreateTaskN\_T$ and $CreateTaskS\_T$, as defined above. Therefore, it is defined by these two subschemas linked by "$\vee$". In the first case, the precondition is that $target?$ is not known to the system and the priority of the new task is lower than or equal to the priority of the running task. The precondition for the second case is that $target?$ is not known to the system and the priority of the new task is greater than that of the running task. Therefore, the precondition for the new schema is only that $target?$ is not known to the system before the operation.

It is worth noticing that the functions in $QReleasingData$ and $MReleasingData$ are auxiliary functions. They are used to help specify the behaviours of queue sending, receiving, and mutex taking operations. In the implementation of FreeRTOS, when a task is woken up after being blocked during the execution of these operations, (for instance, the running task is blocked by sending an item to a full queue) the task continues any unfinished work. However, in the specification, once a task is blocked in a schema, its state will be simply transferred to $blocked$. When it is rescheduled as the running task later, it does not have to continue its unfinished job. It can perform any schema whose precondition is satisfied. Therefore, the functions in $QReleasingData$ and $MReleasingData$ are necessary to distinguish continuing schemas from the others. These functions do not actually represent anything in FreeRTOS. Consequently, preconditions related to them in the specifications are omitted in the tables of Appendix C.

Based on the content of these tables, it is possible to produce code-level annotations for VCC. These preconditions can be used in VCC as the content of requires clauses, "`_(requires ...)`". Further, the postconditions of the schemas can also be transferred into ensures clauses, "`_(ensures ...)`", of the notation of VCC. We have also verified the task-related functions with VCC, which is presented in Chap. 5.

4.7 Some Properties

Finally, there are properties of the specifications that are important for the prover, which have been verified as well. Some of these help to ensure that the specifications have the correct behaviours, the properties of the system are consistent with the API document and source code, etc. Others are used to help Z/Eves prove our model correct. These theorems may seem trivial to the human eye; however, they are particularly helpful for the prover. Therefore, in this section, we present a few of these theorems as examples. Further details can be found in the supplementary material.

1. As described above, in some schemas we need to find the task with the highest priority of all ready tasks. In these cases, ensuring the running task is not a member of the ready tasks is important, otherwise the reschedule algorithm would be chaotic. Moreover, ensuring the running task does not belong to the ready tasks is also important for the prover to prove the related properties of the task. For instance it helps to prove the theorem $TaskProperty6$.

   Theorem 18 (TaskProperty3)

   $$\forall\, Task;\ t : state^{\sim}(\!|\ \{ready\}\ |\!) \bullet t \in tasks \setminus \{running\_task\}$$

2. The variable $tasks$ is used to record tasks known to the system. In other words, if the task is not recorded in $tasks$, it is unknown to the system. This theorem is helpful for the prover to determine the state of this task in $nonexistent$.

Theorem 19 (TaskProperty4)

$$\forall\ Task;\ t : TASK \setminus tasks \bullet state(t) = nonexistent$$

3. As defined by FreeRTOS, the $idle$ task always has the lowest priority. If the priority of a task is greater than 0, this task cannot be the $idle$ task. The command $prove\ by\ reduce$ can be used to prove this.

   Theorem 20 (TaskPriority5)

   $$\forall\ Task;\ t : tasks \mid priority(t) > 0 \bullet t \neq idle$$

4. It is also interesting to check that the behaviour of the operation schemas are properly described. To illustrate this, we select the schema $SuspendTaskS\_T$ to check. This theorem will check that for any proper case of $Task$, after the $SuspendTaskS\_T$ operation, the old running task will be suspended and the new $running\_task$ has the highest priority of the ready tasks. To prove this theorem, the theorem $TaskProperty3$ will be used. Following that, we apply the one-point rule to the condition. The goal can then be proved.

   Theorem 21 (TaskProperty6)

   $$\begin{array}{l}\forall\ Task \mid SuspendTaskS\_T \\ \quad \bullet\ state'(running\_task) = suspended \\ \quad \land\ (\forall\ t : state^{\sim}(\!|\ \{ready\}\ |\!) \\ \quad\quad \bullet\ priority(running\_task') \geq priority(t))\end{array}$$

   proof $[TaskProperty6]$

   $$\begin{array}{l}with\ disabled\ (\Delta\ Task,\ Task)\ prove\ by\ reduce; \\ use\ TaskProperty3[t\ :=\ target!]; \\ instantiate\ t\_\_0\ ==\ t; \\ prove;\end{array}$$

   ■

5. As mentioned in Sect. 2.2, the operator $\oplus$ is commonly used to update the values of a function. However, when the domain of two operands of the operation are disjoint, the effect of this operator is to comprise an union set with the first operand and the second one, which also means that the first function is a subset of the result of the operation.

Theorem 22 (overrideIsAppend)

$$\forall f, g : X \pfun Y \mid \operatorname{dom} f \cap \operatorname{dom} g = \emptyset \bullet f \subseteq f \oplus g$$

6. For a normal function, it is possible that many different elements in the domain of the function can match to one element in the range of the function. In this case, if domain subtraction is applied to this kind of function and there are other elements that can match to the same result as the elements in the subtracted set, the range of the function should not be changed.

   Theorem 23 (ranUnchanged)

$$\forall f : X \pfun Y;\ a : X \mid a \in \operatorname{dom} f \land f(a) \in \operatorname{ran}(\{a\} \ndres f) \bullet \operatorname{ran} f = \operatorname{ran}(\{a\} \ndres f)$$

4.8 Summary

In this chapter, the first section revealed how and why the model is structured in the way described in this chapter. The following four sections described the model in detail, also showing how the model can be extended layer by layer. Finally, the preconditions for the API modelled in the project and some properties of the system were summarised in the last two sections.

In the next chapter, the preconditions, which were calculated in this chapter for task related operations, will be used to develop the VCC annotations and the process of verifying the FreeRTOS implementation (task related API functions) with VCC is described.

Chapter 5

VCC VERIFICATION OF FREERTOS

As mentioned in Chap. 4, the specification defined in the model for FreeRTOS can be used to develop VCC annotations to verify the implementation of FreeRTOS. This chapter illustrates our work on this. Due to time limitations, only task-related functions are verified, to demonstrate the possibility of verifying source code with the abstract model and VCC.

This chapter begins by introducing an overview of VCC and our verification, describing how we organised the specification and the source code. A basic statement of the specification is then given. Finally, annotated functions explain how to use the specification to verify the implementation of FreeRTOS. The full code is not given in the thesis; however, full details can be found in the supplementary material. An explanation of VCC annotations is given, along with a description of the verification.

5.1 Overview

As introduced in Sect. 1.3, VCC is a verifier based on the Z3 prover. It uses annotations, which are ignored by a normal C compiler, to describe the virtual model and the properties of the source code. The following annotations are most frequently used in VCC:

- The most basic annotations in VCC are *assumptions* and *assertions*. Assumptions are used so that the prover considers predicates declared in the assumptions to be logically $true$ and therefore does not attempt to prove them. In contrast, the prover does attempt to prove predicates specified in assertions. However, whether the result is $true$ or $false$, they are also considered as logically $true$ in later proofs.

- It is also possible to define virtual variables and code, which are hidden to an ordinary C compiler, with *ghost* code. This can be used to introduce a virtual model of the specification to the implementation of the C source code.

- Constraints can be defined for data structures (concrete and virtual), which are called *Object Invariants* in VCC.

- An instance of a structure in VCC, called an *object*, has two states, *open* and *closed*. When it is in the *open* state, it is *mutable* and its constraints can be broken. Moreover, if the object is owned by the executing thread, it is writeable for the thread. However, once it is closed, all constraints specified for the data structure have to hold for that instance.

- A function contract can be specified for each function, which is similar to pre- and post-conditions for a schema in Z. This defines the behaviours of a function. VCC assumes the predicates in *requires* annotations in the contract to be $true$ at the beginning of the function and verifies the predicates in *ensures* at the end of the function.

  If the function with the contract is called by another function, the prover does not attempt to prove the called function. It just checks whether the preconditions of the function hold when it is called. If they do, the prover assumes the function is verified, i.e., the postconditions are satisfied.

- In VCC, parameters and global variables are not writeable for a function by default. They have to be included in *writes* annotations to inform the prover

that the function is going to modify the value of the variable. When data is *closed* and owned by the thread, it is called *wrapped*. Even if it is in a *writes* annotation, we also need to *unwrap* it to update the data of the structure.

In addition, there are also other annotations, such as *claim* and *mutable*, etc., that are not as common as the annotations described above. Further details can be found in [21]. They will be briefly described below, where they are used in the verification. It is also worth noting that the concept of hierarchical ownership for variables in VCC exists. For instance, a thread can own several structures and each structure can have its own variables and structures. To update a variable, it has to be included in the writes annotations, opened and owned by the thread.

In theory, we can directly annotate the FreeRTOS source code to verify the implementations. However, as we are only verifying the implementation of task related API functions, due to time limitations, we need to minimise the FreeRTOS source code to only include task-related code. Specifically, we create two files, *vtask.h* and *vtask.c*. In the header file, *vtask.h*, we include some essential predefinitions for verification, such as type definitions for the given sets introduced in the Z specification. This file also contains the functions declarations and their contracts. The file *vtask.c* is an annotated and minimised version of the FreeRTOS source code *task.c*, which includes all definitions of the related API functions. We also modify the API functions to simplify the verification process. We first remove any code not related to the functions verified in the Z specification, such as memory management. Next, we use equivalent code to replace some of the function calls, for instance, `portYIELD_WITHIN_API()`. The most important reason for this decision is that during function calls, the prover may believe that global variables may be changed and it will "forget" information about them. For some cases, it is possible to use claims to inform the prover that this information is not changed during function calls. However, this makes the verification extremely complex. There are also other reasons for different function replacement, which will be explained in detail below, along with the verification.

Figure 5.1: Constraints of tskTCB

```
1  _(dynamic_owns) typedef struct tskTaskControlBlock
2  {
3    xListItem       xGenericListItem;   /*< The list that the state list
         item of a task is reference from denotes the state of that task (
         Ready, Blocked, Suspended ). */
4    xListItem       xEventListItem;   /*< Used to reference a task from an
          event list. */
5    unsigned portBASE_TYPE  uxPriority;     /*< The priority of the task.
         0 is the lowest priority. */
6
7    _(invariant uxPriority < configMAX_PRIORITIES)
8    _(invariant \mine(&xGenericListItem))
9    _(invariant \mine(&xEventListItem))
10 } tskTCB;
11
12 _(ghost typedef tskTCB * TASK;)
```

5.2 Statement Definition

To create the annotations to verify the FreeRTOS implementation with our abstract model, we first need to transform our specification to VCC style annotations. For the task model, the first definition of the specification is the given sets of $TASK$ and $CONTEXT$. Similar to our Z model, we are not interested in the memory management and the detail of context switches, which are mainly implemented by assembly language. We use a pointer of void type (i.e., `void *`) to define $CONTEXT$ in a ghost annotation. However, it is not actually used in the verification. As mentioned in Sect. 1.2.1, a structure called Task Control Block (`tskTCB`) is defined in FreeRTOS to record the properties of a task, such as the priority of the task, general list item (in state lists to represent the state of the task), event list item (in event lists), etc. We use a simplified version of the `tskTCB` structure to define $TASK$ in the verification, which removes memory management-related declarations and some other unrelated fields from the `tskTCB` structure.

As shown in Fig. 5.1, firstly, the priority of a task has to be lower than the maximum priority defined in the configuration file. Then, the list items of each task have to be held by the task. The task cannot be modified by a thread without it being opened, otherwise, the constraints of the system might be broken. Therefore, we state that the pointers for these list items are owned by the `tskTCB` structure, which can later transfer to the thread when it unwraps the task structure. Because of these ownership declarations, it is necessary to mark the structure as "`_(dynamic_owns)`".

Figure 5.2: State and transition in VCC

```
1  _(ghost typedef enum{
2    nonexistent = 0, ready, blocked, suspended, running
3    } STATE;)
4
5  _(ghost \bool transition[5][5];)
6  _(def \bool Trans()
7    {
8      //  nonexistent, ready, blocked, suspended, running  |
9      //-------------------------------------------------+---
10     //{{ \false,     \true, \false,  \false,     \true  } | nonexistent
11     // { \true,      \false,\false,  \true,      \true  } | ready
12     // { \true,      \true, \false,  \true,      \true  } | blocked
13     // { \true,      \true, \false,  \false,     \true  } | suspended
14     // { \true,      \true, \true,   \true,      \false }}| running
15     return
16     transition[0][0] == \false && transition[0][1] == \true &&
17       ... &&
18     transition[4][4] == \false;  // running
19   }
20 )
```

This will request the user to manage the ownership of the object components manually (e.g., manage the ownership of the generic list item, `&xGenericListItem`).

Following this, in Fig. 5.2, an enumerated type, which contains five states of tasks defined in the Z model, is used to translate the free type definition for $STATE$ into VCC. In Z, a set of mappings is used to represent the transition rules, because we know all the elements of this set. Similarly here, the following boolean type two-dimensional array is used to represent this set. Each mapping included in $transition$ can be matched to a boolean $true$ in the array to represent that the transition is valid. Otherwise, if an element of the array is $false$, the represented transition is invalid in this case. It is worth noticing that, in VCC, the curly brackets are overridden to declare object sets. We cannot directly use it to initialise the array. An assistant *logic function* is used to do this. The logic function in VCC is a virtual function, which contains a specification for a given activity. It is normally used to perform some logic checking. As a virtual function, it should not have any operations about memory writes. Specifically, the function `Trans()` returns logical $true$, once the value of each element of $transition$ satisfies the table included in the comments.

Based on these definitions, the virtual structure, `FreeRTOS`, for the base model can be given as below, which organises all the essential elements of verification.

The relationship between these elements and the concrete variables in the original source code can be specified in this virtual structure. Their constraints are also defined here (see Fig. 5.3).

Figure 5.3: FreeRTOS structure

```
1  _(ghost _(dynamic_owns) typedef struct{
2    \bool tasks[TASK];
3    STATE state[TASK];
4    //CONTEXT phys_context;
5    //CONTEXT log_context[TASK];
6
7    \natural priority[TASK];
8
9    //READY
10   _(invariant \forall TASK t; (tasks[t] && xSchedulerRunning != pdFALSE)
         ==>
11       ((((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
             pxReadyTasksLists[((tskTCB *) t)->uxPriority] &&
12        ((tskTCB *) t) != pxCurrentTCB) <==> state[t] == ready))
13   //BLOCKED
14   _(invariant \forall TASK t; (tasks[t] && xSchedulerRunning != pdFALSE)
         ==>
15       ((((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
             xDelayedTaskList1 ||
16         ((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
              xDelayedTaskList2) <==> state[t] == blocked))
17   //SUSPENDED
18   _(invariant \forall TASK t; (tasks[t] && xSchedulerRunning != pdFALSE)
         ==>
19       (((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
             xSuspendedTaskList <==> state[t] == suspended))
20   //RUNNING
21   _(invariant \forall TASK t; (tasks[t] && xSchedulerRunning != pdFALSE)
         ==>
22       (t == (TASK) pxCurrentTCB <==> state[t] == running))
23   _(invariant ((xList *) pxCurrentTCB->xGenericListItem.pvContainer) ==
         &pxReadyTasksLists[pxCurrentTCB->uxPriority])
24   //NONEXISTENT
25   _(invariant \forall TASK t; (t->\closed && xSchedulerRunning !=
         pdFALSE) ==>
26       ((((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
             xTasksWaitingTermination ||
27         t == NULL) <==> state[t] == nonexistent))
28
29   _(invariant \forall TASK t; xSchedulerRunning != pdFALSE ==> (state[t]
         != nonexistent <==> tasks[t]))
30   _(invariant \forall TASK t; tasks[t] ==> \mine(t))
31   _(invariant \forall TASK t; tasks[t] ==> t->\closed)
32   _(invariant \forall TASK t; tasks[t] ==> priority[t] == t->uxPriority)
33   _(invariant \forall TASK t; tasks[t] ==> state[t] <= 4)
34   _(invariant \mine(\embedding(& xIdleTaskHandle)))
35   _(invariant \mine(\embedding(& pxCurrentTCB)))
36   _(invariant xSchedulerRunning != pdFALSE ==> (tasks[xIdleTaskHandle]
         && tasks[pxCurrentTCB]))
37   _(invariant xSchedulerRunning != pdFALSE && xIdleTaskHandle != NULL
         ==> priority[xIdleTaskHandle] == 0)
38   _(invariant \forall TASK t; xSchedulerRunning != pdFALSE && state[t]
         == ready ==> priority[pxCurrentTCB] >= priority[t])
39
40   } * FreeRTOS;)
```

This structure translates the schema $Task$ into a VCC style base model. The map `tasks` from `TASK` to `\bool` in Ln. 2 is used to represent $tasks$ in the Z model. In VCC, map type is a type similar to an array, which can only be used in ghost

code. It maps a value of the type in the square bracket to a value of the type before the variable name. In this case, `\bool tasks[TASK]` maps each task to a boolean type value to indicate whether the task is known by the system or not. Similarly, a map from `TASK` to `STATE` and a map from `TASK` to natural numbers are used to represent $state$ and $priority$, respectively. As we mentioned above, the context related variables are not used in the verification. They are declared, but commented out, which has no effect on our model.

After the declarations of the virtual variables, the links between them and the real variables are specified. Specifically, lines 9 to 27 in Fig. 5.3 show the relationship between the virtual state in the specification to the real state lists in FreeRTOS. For instance, FreeRTOS declares an array of `tskTCB` lists to store ready tasks. An invariant (Ln. 9 to 12) is used to specify this, which defines that all the tasks in a list of `pxReadyTasksLists` are exactly those that are in a ready state, i.e., `state[t] == ready`.

Following that, line 29 translates a constraint from the $tasks$ schema. It states that all tasks which are not in a `nonexistent` state are known by the system (i.e., they map to $true$ in `tasks`). However, if a task is known to the system, we can also say it belongs to the structure `FreeRTOS` (Ln. 30), as this structure describes the base of the whole system and is the root of the hierarchy ownership tree. When a thread unwraps the `FreeRTOS` structure from its *wrapped* state, the thread automatically obtains ownership of all tasks, which makes these tasks in the *wrapped* state at the moment. Otherwise, to verify the properties of the tasks for a function, it would have to have all the tasks in the parameter list of the function, which is not possible. A binding from the virtual priority to the real priority of each task is stated in Ln. 32.

Next, the handlers of the `idle` task and `pxCurrent` task are declared as global variables in FreeRTOS. If we want to specify the properties of these two handlers, we need to transfer their ownership to the `FreeRTOS` structure. To do this, we first need to transfer ownership of their container to `FreeRTOS`. In VCC, all global fields are included in a virtual container. Due to their definition in Ln. 30, setting

their entry in `tasks` to $true$ transfers ownership to the `FreeRTOS` structure. The constraint that the priority of `idle` task has to be 0 can then be specified. The last invariant shows that the priority of the running task is the highest of all ready tasks, which is also the last constraint of the $Task$ schema in Z.

Finally, another logic function, `excList()` (details of which can be found from the supplementary material), is given to ensure that the state lists for tasks are exclusive (i.e., one list cannot be used as more than one state).

5.2.1 Translation from Z to VCC

In summary, to translate types from Z to VCC, the following rules can be considered as a guide.

- Given sets in Z can be translated to point types or structures defined in C, e.g. $TASK$ is translated to `tskTCB *`.
- Generally, functions can be translated into two ways, (a) As functions are sets of mappings in Z, they can be naturally translated to maps in VCC, e.g. the function $priority$ is translated to `\natural priority[TASK]`; (b) If a function is defined from a given set which has been translated as a structure, to another type, the value of this function can be defined as a ghost field in the structure, e.g. the function $priority$ can also be translated as a ghost field in the structure `tskTCB`. We chose the former way to translate $priority$, as it is closer to our Z specification.
- Sets in Z can be understood as power sets of their own type. Therefore, in VCC a set can be defined by a boolean type map that maps each element of its type into a boolean type value to indicate whether the element is a member of the set e.g. $tasks$ is translated to `\bool tasks[TASK]`.
- Simple free types can be translated as enumerated types e.g. $STATE$ defined in Fig. 5.2.
- Finally, static functions can be defined as two-dimensional arrays of Booleans, e.g. $transition$ is represented by `\bool transition[5][5]`.

VCC provides plenty of operators for the ghost fields and logical operations, including quantifiers ($\forall$ and $\exists$), implication, etc. Once all the variables in our Z specification have been translated to corresponding VCC variables or C variables, we can use operators defined in VCC and C for these types of variables to represent predicates used in the Z specification. For example,

- For functions, which are translated as mappings, we can simply apply the function to an element in the domain to obtain the value of the function, e.g. `state[t]` used in Ln. 12 of Fig. 5.3.
- Similarly, for sets in Z,
  - we can calculate the result of a map applied to an element to check whether the element is in the set, e.g. `tasks[t] == \true`;
  - we can also set the value of a map for an element to `\true` or `\false` to add or remove the element to or from the set respectively.
- As VCC supports quantifiers, we can directly translate such predicates from Z specification to VCC annotations, e.g. we use plenty of quantifier `\forall` in Fig. 5.3.

More detailed explanation will be provided during the explanation of each API functions in following sections.

Following this, we can use the preconditions and postconditions verified in the Z model to verify the implementation of FreeRTOS task API functions.

5.3 Creating Tasks

To verify the API function for creating tasks in FreeRTOS, `xTaskGenericCreate`, we first append two ghost parameters, `FreeRTOS` and `newTask`, to the function parameter list. These represent the virtual model and the new task created by the function respectively. The contract of the API function can then be specified as in Fig. 5.4. The first annotation, `_(updates FreeRTOS)`, is used to tell the prover that the structure is *wrapped* before and after the operation. It may be modified by the operation (i.e., `FreeRTOS` is writeable and in a *writes* clause). It is

actually equivalent to `_(requires \wrapped(FreeRTOS))`, `_(writes FreeRTOS)` and `_(ensures \wrapped(FreeRTOS))`.

As our specification in Z assumes that the scheduler is always executing (see Sect. 4.2.1), it is stated for VCC verification as well. Following this, the logic function `excList()` is used to ensure that all items in the state list are exclusive when the API function starts. Because the value of `newTask` and `pxCreatedTask` will be updated to the handler of the created task, they have to be included in the writes clause of the VCC annotation. In addition, the postconditions can be specified with ensures clauses.

In the Z specification, we separate the creating operation into two cases, normal and rescheduling. The postconditions for these need to be mixed together in the same way as the postconditions for the API function. VCC provides `\old()` function to obtain the pre-state of a variable, which is used here to refer to the old value of `pxCurrentTCB`. Then the priority of the old running task can be accessed. Comparing the new priority of the new task and the priority of the old running task, the difference between the two cases can be specified separately. In the normal case of task creation (see Ln. 9 to 11), the priority of the new task is less or equal to the priority of the old running task. The created task is set to the ready state. Otherwise (see Ln. 12 to 14), it requests rescheduling and replaces the value of `pxCurrentTCB`. The state of the created task and the old running task should also be set to `running` and `ready`, respectively. For both cases, the priority in the parameter list should be assigned to the new task.

It is worth noting that in the Z model, we do not restrict the maximum value of the priority. In FreeRTOS, however, the maximum value of priority, `configMAX_PRIORITIES`, is defined in the configuration file. If the new priority is not less than the maximum priority, it is set to the maximum priority of the system (i.e., `configMAX_PRIORITIES-1`).

The only precondition for creating tasks in the Z model is that the created task is not in the system before the operation. As it is newly created during the operation,

Figure 5.4: Contract for creating tasks

```
1  signed portBASE_TYPE xTaskGenericCreate( pdTASK_CODE pxTaskCode, const
       signed char * const pcName, unsigned short usStackDepth, void *
       pvParameters, unsigned portBASE_TYPE uxPriority, xTaskHandle *
       pxCreatedTask, portSTACK_TYPE *puxStackBuffer, const xMemoryRegion
       * const xRegions _(ghost FRTOS FreeRTOS) _(ghost TASK *newTask) )
2
3  _(updates FreeRTOS)
4  _(requires xSchedulerRunning == pdTRUE)
5  _(requires excList())
6
7  _(writes newTask, pxCreatedTask)
8
9  _(ensures \result == pdPASS ==>
10   uxPriority <= \old(pxCurrentTCB)->uxPriority ==>
11     (FreeRTOS->state[*newTask]) == ready)
12 _(ensures \result == pdPASS ==>
13   uxPriority > \old(pxCurrentTCB)->uxPriority ==>
14     (FreeRTOS->state[(TASK)\old(pxCurrentTCB)]) == ready && (FreeRTOS->
           state[(TASK)pxCurrentTCB]) == running)
15 _(ensures \result == pdPASS ==> FreeRTOS->priority[*newTask] ==
16     (\natural)(uxPriority < configMAX_PRIORITIES ? uxPriority :
           configMAX_PRIORITIES - ( unsigned portBASE_TYPE ) 1U))
17 _(ensures \result == pdPASS ==> pxCurrentTCB->\closed)
18 _(ensures \result == pdPASS ==> \fresh(*newTask))
19 _(ensures \forall TASK t; (FreeRTOS->tasks[t] &&
20  \old(FreeRTOS->state[t]) != FreeRTOS->state[t]) ==>
21     transition[\old(FreeRTOS->state[t])][FreeRTOS->state[t]])
```

it is not easy to describe with requires clauses; therefore, it is stated in the post-condition instead. The function, `\fresh()` is provided by VCC to indicate that the object is freshly allocated. Finally, the last predicate requests VCC to check that the state transitions obey `transition`.

To improve the efficiency of the prover, VCC does not make forward inferences from the precondition by default. For instance, in the precondition, we state that `FreeRTOS` is *wrapped*, which means all the constraints have to hold at the beginning of the function. In the constraints of the `FreeRTOS` structure, we state that `pxCurrentTCB` is owned by `FreeRTOS` and all the tasks owned by `FreeRTOS` have to be closed. From these definitions, we can easily conclude that `pxCurrentTCB` has to be in the *closed* state, when `FreeRTOS` is *wrapped*. However, VCC cannot obtain this result automatically. Therefore, the script in Fig. 5.5 is used to show that `pxCurrentTCB` is *closed* at the beginning along with some other properties which may be helpful for later verification. After the declaration of some local variables, the first statement calls the function `prvInitialiseTCBVariables`, which allocates memory for the new task and its stack. To simplify the verification, we replaced it with `malloc`. An annotation is added here to make sure that the prover

Figure 5.5: Creating tasks pre-verification

```
1  _(assert \wrapped(FreeRTOS))
2  _(assert \inv(FreeRTOS))
3  _(assert xSchedulerRunning == pdTRUE)
4  _(assert FreeRTOS->tasks[pxCurrentTCB])
5  _(assert pxCurrentTCB \in FreeRTOS->\owns)
6
7  _(assert \forall TASK t; FreeRTOS->tasks[t] ==> \inv(t))
8  _(assert \forall TASK t; FreeRTOS->tasks[t] ==> FreeRTOS->priority[t] <
       configMAX_PRIORITIES)
9
10 _(assert \forall TASK t; FreeRTOS->tasks[t] ==> FreeRTOS->state[t] <= 4)
11 _(assert pxCurrentTCB->\closed)
12
13 signed portBASE_TYPE xReturn;
14 tskTCB * pxNewTCB;
15
16 \\pxNewTCB = prvInitialiseTCBVariables( usStackDepth, puxStackBuffer );
17 pxNewTCB = (tskTCB *) malloc(sizeof(tskTCB));
18 _(assert pxNewTCB != pxCurrentTCB)
```

ensures that the handler for the new task is not equal to `pxCurrentTCB`, otherwise, when it is updated, the prover may confuse these two tasks as they refer to the same task.

As shown in Fig. 5.6, when `pxNewTCB` is successfully assigned memory, two assertions (Ln. 2 and 3) are added to help the prover ensure that `pxNewTCB` is successfully and newly allocated by the function. It is then necessary to complete the details of the created task. As we only focus on the functions we specified in the Z model, we remove all unrelated code and only keep functions related to the priority and the generic list item of the new task, which sets the new priority for the task and places it in the proper position in the ready task lists. The task is then physically created. The rest of the creation function manages the system state for the new task. In VCC, before verifying system properties related to `pxNewTCB`, the structure has to be wrapped in advance. To do this, the ownership of all `pxNewTCB` components has to be taken by the structure. However, for the new created task, the thread keeps the ownership of `pxNewTCB` and all its components, i.e., the generic and event list items, which need to be transferred to `pxNewTCB`. Two ghost statements in Ln. 13 and 14 are used to perform this and then the statement `_(wrap pxNewTCB)` is used to wrap it. In the critical section, the original source code manages the system states according to the properties of the new tasks, for instance, the maximum priority of existing tasks, the total

number of the tasks in the system, etc. However, they are not included in our reduced version. The variable `xReturn` is then set to `pdPASS` to indicate that the new task has been successfully created and added to the system. In cases of failure to allocate memory to `pxNewTCB`, the function does nothing but set the value of `xReturn` to `errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY`. In this case, an assertion is needed to remind the prover that the invariants of `FreeRTOS` still hold at that time.

Figure 5.6: Creating tasks verification part-1

```
1  if ( pxNewTCB != NULL ) {
2    _(assert pxNewTCB)
3    _(assert \fresh(pxNewTCB))
4
5    //prvInitialiseTCBVariables( pxNewTCB, pcName, uxPriority, xRegions,
         usStackDepth );
6    pxNewTCB->uxPriority = (uxPriority < configMAX_PRIORITIES ? uxPriority
         : configMAX_PRIORITIES - ( unsigned portBASE_TYPE ) 1U);
7    pxNewTCB->xGenericListItem.pvContainer = &pxReadyTasksLists[pxNewTCB->
         uxPriority];
8    _(assert \writable(&(pxNewTCB->xGenericListItem)))
9    _(assert \writable(&(pxNewTCB->xEventListItem)))
10   _(wrap &(pxNewTCB->xGenericListItem))
11   _(wrap &(pxNewTCB->xEventListItem))
12
13   _(ghost pxNewTCB->\owns = (\objset) {&(pxNewTCB->xGenericListItem)})
14   _(ghost pxNewTCB->\owns += &(pxNewTCB->xEventListItem))
15   _(assert !(FreeRTOS \in pxNewTCB->\owns))
16   _(wrap pxNewTCB)
17
18   if ( ( void * ) pxCreatedTask != NULL ) {
19     *pxCreatedTask = ( xTaskHandle ) pxNewTCB;
20   }
21   taskENTER_CRITICAL();
22   { //...
23     xReturn = pdPASS;
24   }
25   taskEXIT_CRITICAL();
26 } else {
27   _(assert \inv(FreeRTOS))
28   xReturn = errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY;
29 }
```

When `xReturn` is set to `pdPASS`, the system needs to check whether rescheduling is requested and the ghost code for managing system states is placed here, as shown in Fig. 5.7. Firstly, before the system checks for rescheduling, we unwrap the `FreeRTOS`, set the `priority` of `pxNewTCB`, temporarily set the state of `pxNewTCB` to `ready`, add it to `tasks` and transfer its ownership to `FreeRTOS`. Finally, we assign `pxNewTCB` to the virtual parameter `* newTask`. Some assertions are used here to ensure that the assignments of the virtual code work. The system then checks the relationship between the priority for the new task and the priority for the running

task to determine whether the system needs to reschedule or not.

As a global volatile variable, pxCurrentTCB can be accessed and updated by a thread in VCC which does not hold its ownership. To access pxCurrentTCB, an *atomic_read* annotation is applied to the container of pxCurrentTCB, which is \embedding(&pxCurrentTCB) in this case. When the priority of pxCurrentTCB is lower than the new priority, the system needs to reschedule. We also modify the state of pxNewTCB and pxCurrentTCB to running and ready, respectively. A system function call to portYIELD_WITHIN_API is used to perform rescheduling, which is replaced by directly assigning the handler of the new task to pxCurrentTCB in our reduced version.

Similar to accessing pxCurrentTCB, in order to modify it, we need to apply the *atomic* annotation to its container. It is worth noting that during the atomic operation, information about global variable FreeRTOS is lost. Ideally, this can be handled by the *claim* annotation, which claims that the state of FreeRTOS is kept unchanged. However, this complicates the verification and due to the time limits of the project, we simply assert that the invariant of FreeRTOS holds before the atomic operation and assume them after the operation. In addition to these, some assertions are used to ensure that the properties for FreeRTOS are well maintained. It can then be wrapped. Finally, the assumption of the logic function, Trans(), is used to set the value of the array transition.

5.4 Deleting Tasks

Similar to creating tasks, to prove the API function for deleting tasks, vTaskDelete, two extra ghost parameters are appended, FreeRTOS and topReady. As shown in Fig. 5.8, the first four annotations in the function contract are similar to used in creating tasks. They describe the basic properties of the system. In addition, the preconditions specific to deleting tasks can be defined following our Z specification, which states the target task has to be a member of $tasks$ but not $idle$ and the $topReady!$ is the highest priority ready task in the system. In the source code, the target task is named pxTaskToDelete. We first state that at the beginning of

Figure 5.7: Creating tasks verification part-2

```
1  if( xReturn == pdPASS )
2  {
3    _(unwrapping FreeRTOS) {
4      _(assert \inv(FreeRTOS))
5
6      _(ghost {
7        FreeRTOS->priority[(TASK) pxNewTCB] = pxNewTCB->uxPriority;
8        FreeRTOS->state[(TASK) pxNewTCB] = ready;
9        FreeRTOS->tasks[(TASK) pxNewTCB] = \true;
10       FreeRTOS->\owns += pxNewTCB;
11       *newTask = pxNewTCB;
12     })
13
14     _(assert FreeRTOS->state[(TASK) pxNewTCB] == ready)
15     _(assert FreeRTOS->state[(TASK) pxCurrentTCB] == running)
16     _(assert FreeRTOS->priority[(TASK) pxNewTCB] == pxNewTCB->uxPriority
           )
17     _(assert \forall TASK t; (FreeRTOS->tasks[t] && ((tskTCB *)t !=
           pxNewTCB) && xSchedulerRunning != pdFALSE) ==>
18         ((((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
               pxReadyTasksLists[((tskTCB *) t)->uxPriority] &&
19           ((tskTCB *) t) != pxCurrentTCB) <==> FreeRTOS->state[t] ==
                ready))
20
21     ...
22
23     if( _(atomic_read \embedding(&pxCurrentTCB))
24         pxCurrentTCB->uxPriority < uxPriority )
25     {
26       _(ghost {
27         FreeRTOS->state[(TASK) pxCurrentTCB] = ready;
28         FreeRTOS->state[(TASK) pxNewTCB] = running;
29         })
30       _(assert FreeRTOS->state[(TASK) \old(pxCurrentTCB)] == ready)
31       //portYIELD_WITHIN_API();
32       _(atomic \embedding(&pxCurrentTCB)){
33         pxCurrentTCB = pxNewTCB;
34         _(bump_volatile_version \embedding(&pxCurrentTCB))
35       }
36     }
37
38     _(assert \old(pxCurrentTCB)->uxPriority >= uxPriority ==> (FreeRTOS
           ->state[(TASK) pxNewTCB] == ready && FreeRTOS->state[(TASK)
           pxCurrentTCB] == running))
39     _(assert \old(pxCurrentTCB)->uxPriority < uxPriority ==>
40       (FreeRTOS->state[(TASK) pxNewTCB] == running &&
41         pxNewTCB == pxCurrentTCB &&
42         FreeRTOS->state[(TASK) \old(pxCurrentTCB)] == ready
43       )
44     )
45     _(assert FreeRTOS->priority[(TASK) pxNewTCB] == pxNewTCB->uxPriority
           )
46     _(assert FreeRTOS->priority[(TASK) \old(pxCurrentTCB)] == \old(
           pxCurrentTCB)->uxPriority)
47     _(assert FreeRTOS->tasks[(TASK) \old(pxCurrentTCB)])
48
49     _(assume \forall TASK t; (FreeRTOS->tasks[t] && ((tskTCB *)t !=
           pxNewTCB) && xSchedulerRunning != pdFALSE) ==>
50         ((((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
               pxReadyTasksLists[((tskTCB *) t)->uxPriority] &&
51           ((tskTCB *) t) != pxCurrentTCB) <==> FreeRTOS->state[t] ==
                ready))
52
53     ...
54
55     _(assert \forall TASK t; FreeRTOS->tasks[t] ==> t->\closed)
56   }//wrapping FreeRTOS
57 }
58 _(assume Trans())
59 return xReturn;
```

Figure 5.8: Contract for deleting tasks

```
1  _(updates FreeRTOS)
2  _(requires \mutable(&xSchedulerRunning))
3  _(requires xSchedulerRunning == pdTRUE)
4  _(requires excList())
5
6  _(requires FreeRTOS->tasks[(TASK) pxTaskToDelete])
7  _(requires (tskTCB *)pxTaskToDelete != (tskTCB *)xIdleTaskHandle)
8
9  _(requires FreeRTOS->tasks[topReady])
10 _(requires FreeRTOS->state[topReady] == ready)
11 _(requires \forall TASK rts;
12     (FreeRTOS->tasks[rts] && FreeRTOS->state[rts] == ready)
13         ==> FreeRTOS->priority[topReady] >= FreeRTOS->priority[rts])
14 _(requires topReady != (TASK) pxTaskToDelete)
15 _(requires topReady != (TASK) pxCurrentTCB)
16
17 _(ensures ((TASK) pxTaskToDelete)->\closed)
18 _(ensures pxTaskToDelete != NULL ==> ! FreeRTOS->tasks[(TASK)
       pxTaskToDelete])
19 _(ensures pxTaskToDelete == NULL ==> ! FreeRTOS->tasks[(TASK) \old(
       pxCurrentTCB)])
20 _(ensures pxTaskToDelete == NULL ==> (TASK) pxCurrentTCB == topReady)
21 _(ensures \forall TASK t; (FreeRTOS->tasks[t] &&
22   \old(FreeRTOS->state[t]) != FreeRTOS->state[t]) ==>
23     transition[\old(FreeRTOS->state[t])][FreeRTOS->state[t]])
```

the function, `pxTaskToDelete` has to be in `tasks` and not equal to the handler of the idle task, `xIdleTaskHandle`. Then, `topReady` has to be in `tasks` with a state of `ready` and have the highest priority of the ready tasks. Moreover, it is also important to state that the `topReady` must not be the same as the running task or the task to be deleted. Otherwise, the deleted task can be set to the new running task after the operation, which is not sensible.

Compared to creating tasks, the postconditions for deleting tasks are quite simple. They ensure that the target task is removed from the system by checking the value of the `tasks`. When the deleted task is the old running task, `topReady` is used to replace the running task. It needs to be noted that in FreeRTOS, if the handler of the deleted task in parameter list is `NULL`, the running task will be deleted. Like creating tasks, the last postcondition ensures that the state transfer for all tasks follows the restriction defined by `transition`.

Again, similar to creating tasks, we firstly need to ensure the prover retains the necessary information about the system. Before entering the critical section, we unwrap `FreeRTOS` (see Ln. 347 on Page. 348), as the deleted task is already owned by `FreeRTOS`. To delete it, `FreeRTOS` has to be opened. As mentioned above,

FreeRTOS defines a NULL value for the target task, which indicates that the target task is the running task. This is because the macro prvGetTCBFromHandle is used to obtain the control block for a task from its handle.

Meanwhile, the macro returns the control block of the running task, while the handle is NULL. To simplify the code, FreeRTOS sets the value of the parameter pxTaskToDelete to NULL, if it is equal to pxCurrentTCB. It then assigns the task obtained from pxTaskToDelete to pxTCB, which represents the task to be deleted afterwards. These two steps require access to the global volatile variable, pxCurrentTCB. The atomic read declaration is needed for this. As the virtual structure FreeRTOS has been unwrapped, the ownership of tasks in the system is now transferred to the thread and hence they are now wrapped. Assertion (in Ln. 370 on Page. 349) is used to ensure that pxTCB is wrapped at that time, and can then be unwrapped.

The function, uxListRemove, is used to remove pxTCB's generic list item from the system. Furthermore, it can be added to the list xTasksWaitingTermination to mark that it needs to be removed. Then the idle task will make the deletion when it is executed. It is also necessary to remove the event list of the target task, if it is in any event list. Three assertions are used to ensure these operations perform properly. Following this, a piece of ghost virtual code (between Ln. 403 and Ln. 406 on Page. 349) is used to maintain the system state. The state of pxTCB is set to nonexistent, and the value of pxTCB in tasks is set to \false as well. Exiting the critical session, the API function checks whether the running task has been deleted. If it has been, rescheduling is required. Specifications and code similar to creating task are used here to perform rescheduling. Again, due to the atomic operation, we assert and assume the necessary properties of FreeRTOS before and after atomic access. It can then be wrapped. Finally, assumption of Trans is used to initialise the array transition, which can be used for the postcondition check, at a later point.

5.5 Getting and Setting Priority

The API function, `uxTaskPriorityGet`, is the simplest to verify (see Ln. 508-539 on Page. 351). It keeps everything unchanged and returns the priority of the target task. The precondition states that the target task (i.e., `pxTask`) is known to the system and other general system preconditions, such as the one to create tasks. The postcondition ensures that the returned value, `\result`, equals `FreeRTOS->priority[(TASK) pxTask]`.

To set the priority of a task, the API function `vTaskPrioritySet` can be used. To verify this, similar to deleting a task, the extra virtual parameters `FreeRTOS` and `topReady` are appended. However, in this case, the details of `topReady` remain undefined at the beginning. The key reason for this is that `vTaskPrioritySet` requests a system reschedule, when the priority of the running task is reduced. In this case, the running task may still be scheduled as the new running task again. Therefore, the definitions for the details of `topReady` are specified when rescheduling is required.

As well as the four general preconditions, there are three extra annotations for the precondition (see Ln. 551-560 on Page. 351). These state that `topReady` and `pxTask` are in the system and that when the target task is the idle task, the priority has to be 0. The postconditions for the `vTaskPrioritySet` (see Fig. 5.9) first state that the target task has to be closed and known by the system. The key feature for this API function is then to update the priority of the target task. An annotation is used to ensure that this has been done successfully. In our specification, there are two cases where the system needs to be rescheduled, (a) when the target task is in the ready state and obtains a higher priority than the running task; and (b) when the priority of the running task is reduced to less than one of the ready tasks. The following two predicates are specified to check that in these two cases the system is rescheduled properly. Finally, it is verified that the state transitions are valid for this operation.

Similarly, some assertions are used to show basic properties. After the dec-

Figure 5.9: Postconditions for priority setting

```
1  _(ensures ((TASK) pxTask)->\closed)
2  _(ensures FreeRTOS->tasks[(TASK) pxTask])
3  _(ensures FreeRTOS->priority[(TASK) pxTask] ==
4      (\natural)(uxNewPriority < configMAX_PRIORITIES ? uxNewPriority :
           configMAX_PRIORITIES - ( unsigned portBASE_TYPE ) 1U))
5  _(ensures (FreeRTOS->state[(TASK) pxTask] == ready && FreeRTOS->priority
       [(TASK) pxTask] > \old(pxCurrentTCB)->uxPriority) ==>
6      (FreeRTOS->state[(TASK) \old(pxCurrentTCB)] == ready && FreeRTOS->
           state[(TASK) pxTask] == running)
7  )
8  _(ensures ((pxTask == NULL || (tskTCB *) pxTask == \old(pxCurrentTCB))
       && !(
9          \forall TASK t; FreeRTOS->state[t] == ready ==> FreeRTOS->
               priority[(TASK) pxTask] >= FreeRTOS->priority[t]
10     )) ==> (FreeRTOS->state[(TASK) \old(pxCurrentTCB)] == ready &&
11             FreeRTOS->state[(TASK) pxCurrentTCB] == running &&
12             \old(FreeRTOS->state[(TASK) pxCurrentTCB]) == ready)
13 )
14 _(ensures \forall TASK t; (FreeRTOS->tasks[t] && \old(FreeRTOS->state[t
       ]) != FreeRTOS->state[t]) ==> transition[\old(FreeRTOS->state[t])][
       FreeRTOS->state[t]])
```

laration, the API function first checks whether the new priority is less than `configMAX_PRIORITIES` (see Ln. 594 on Page. 352). If it is not, it is set to the maximum priority. An assertion is added to ensure this. It then enters the critical session to update the priority of the target task and manage the system states. In this case, the `FreeRTOS` structure needs to be unwrapped. It transfers the ownership of `pxTCB` to the threads. As in the case of deleting tasks, if `pxTask` is equal to `pxCurrentTCB`, it is set to `NULL`. With the macro `prvGetTCBFromHandle` the task control block for the target task is obtained and assigned to `pxTCB`. Atomic read annotations are also used for reading from `pxCurrentTCB`. The functions use a local variable, `uxCurrentPriority` to record the old priority of the target task. After obtaining the control block for the target task, its priority is also accessible and assigned to `uxCurrentPriority`.

Afterwards, the local variable `xYieldRequired` is set to `pdTRUE` to indicate that rescheduling is requested, according to the relation between the old and new priority of the target task (see Ln. 641-663 on Page. 353). Specifically, if it is increased and the target task is not the running task, or the target task is the running task and its priority is decreased, it indicates that there may be a ready task that has a higher priority than the running task. `xYieldRequired` is set. Following this, the function starts to update the priority of `pxTCB` and manage the related system variables. At

this point, some assertions are inserted into the source code to verify that pxTCB is wrapped. We then unwrap it to allow the thread to update the priority of pxTCB. The priority of the target task is also updated to the mapping priority of the virtual model. Furthermore, if a task is in the ready or running state, its generic list item should be placed in one of the ready lists, according to its priority. Therefore, the target task has to be replaced in the new list based on its new priority, which helps the system perform rescheduling correctly.

During this process, we first need to inform the prover that while pxTCB's generic list item is placed in one of pxReadyTasksLists, it is in the ready or running state (see Ln. 702 on Page. 354). Moreover, to replace its generic list item to the new position of the pxReadyTasksLists, &(pxTCB->xGenericListItem) is unwrapped.

Finally, assertions are inserted to verify that after this process, the target task is still in the mapping tasks and its generic list item is replaced in the correct position. The last part of the function is used to reschedule the system, if xYieldRequired is set to pdTRUE (see Ln. 726 on Page. 354). Similar to the previous API functions, portYIELD_WITHIN_API is used to perform this. It is replaced with ghost code as well. The assertions and assumptions are used to ensure the operation works. The only thing different here is that we need to specify the details of topReady above the ghost code. Because the API function requests rescheduling when the priority of the running task is reduced, it can still have the highest priority compared to the other ready tasks. Therefore, topReady can be pxCurrentTCB for this API function. Further, it has to have the highest priority compared to the other ready tasks. If the running task still has the highest priority, it is set to topReady. Like the previous API functions, the atomic access has lost some of the information about FreeRTOS, so we simply assume that they hold. It can then be wrapped before exiting the critical session. Finally, the assumption for Trans() is inserted.

5.6 Suspending and Resuming Tasks

In the FreeRTOS task model in Z, we also verify the API functions for suspending (see Page. 355-358) and resuming (see Page. 358-359) tasks, which are sim-

ilar to deleting and creating tasks. The key difference between suspending and deleting tasks is the destiny of the target task, which are the suspended state and the non-existent state respectively. Due to space limitations they are not presented here, but can be found from the supplementary material.

5.7 Summary

This chapter firstly introduced VCC and the overview of our verification with VCC. The rest of the chapter showed our verification for task related functions in the FreeRTOS implementation. The work described in this chapter shows the possibility of verifying executable code with an abstract specification and code verifier.

In the next chapter, the FreeRTOS model described in Chap. 4 will be extended to suit a multi-core platform.

Chapter 6

EXTENSION FOR MULTI-CORE

This chapter extends the model described in Chap. 4 to a multi-core platform. It also follows the structure adapted in Chap. 4, which develops the $Task$, $Queue$, $Time$ *and* $Mutex$ *models. Due to time limitations, we were only able to extend the FreeRTOS specification to a multi-core platform; the proof of the consistency of the model could not be completed and is part of our future work. In addition, it was not possible to show all details of the model in the thesis. Full details can be found from the supplementary material.*

6.1 Overview

To migrate FreeRTOS to a multi-core platform, it is important to find a new scheduling policy for tasks. Similar to the model for FreeRTOS, this is an abstract model. Therefore, one option is to leave the scheduling algorithm as nondeterministic. It could then be refined as required at the refinement stage. However, because the system described is an extension of FreeRTOS, we decided to extend the scheduling algorithm for FreeRTOS to a multi-core algorithm. This provides benefits for the development in later stages. For instance, if we apply the promotion technique to develop the new specification of the multi-core model[1], the specification for FreeRTOS can be reused. As mentioned previously, in FreeRTOS, tasks are

[1] We produced another version of the task model with promotion, which is shown in Sect. 8.2.

scheduled based on their priority. On a multi-core platform, tasks execute in parallel. Therefore, rather than time-slicing a single processor, a simple scheduling policy is used to extend the notion of a highest-priority ready task to a set of highest-priority tasks sufficient for the number of cores available. This is called *global scheduling*. However, it is known to be less efficient than *partitioned scheduling*, where each task is bound to a specified core and scheduling occurs within the core [73]. The main reason for this is the cost of migrating tasks from one core to another. We adopted a priority-based partitioned scheduling policy.

As this was the first attempt, we aimed to keep everything as simple as possible. Therefore, we avoided specifying details about the architecture of the multi-core platform, memory and interrupt management. Following the structure of the FreeRTOS model, we divided our specification for the multi-core platform into four major parts: $Task$, $Queue$, $Time$ and $Mutex$. The well-definedness of these models was checked by Z/Eves. Furthermore, they can also be animated by ProZ.

6.2 Task Model

This is the core part of the specification. It includes operations for task management, such as creating tasks and deleting tasks, *etc.* Similar to Sect. 4.2.1, we also need to define some basic statements that are used in the specification.

6.2.1 Basic Statements

As well as the given sets defined in Sect. 4.2.1, we introduce another given set, $CORE$, to represent the cores.

$$[CONTEXT, TASK, CORE]$$

The constant, $bare\_context$ represents the initial state of the processors, similar to the FreeRTOS model. The multi-core operating system may have multiple `idle` tasks, one for each core; therefore, the constant $idle$ need to be updated to $idles$, a finite set of tasks. The last constant $cores$ represents all the cores available for the system. As each core can have only one `idle` task, the number for cores and `idle` tasks should be the same.

$$
\begin{array}{|l}
bare\_context : CONTEXT \\
idles : \mathbb{F}\ TASK \\
cores : \mathbb{F}\ CORE \\
\hline
\#cores = \#idles \\
cores \neq \emptyset
\end{array}
$$

As in FreeRTOS, we defined five states for tasks in our system. Their state transitions should also obey $transition$ defined in Sect. 4.2.1.

Similar to the FreeRTOS model, we also have four sub-state schemas to describe all essential properties of tasks in the system: $TaskData$, $StateData$, $ContextData$ and $PrioData$. $TaskData$ includes the most basic properties of the system. It first introduces a finite set $tasks$ to represent all the tasks in the system. As a multi-core system, there is more than one running task in the system: each core must always have a running task and the running task is uniquely run on one core. Therefore, the definition of the running task is updated to an injective total function, $running\_tasks$, which shows the relation between cores and their running tasks. Finally, a partial surjective function, $executable$, is given to record the relation between each task and its core. As we adopted a partitioned scheduling policy in our system, this information is essential for scheduling. To guarantee the properties mentioned above, four constraints are specified.

$$
\begin{array}{|l}
\hline
TaskData \\
\hline
tasks : \mathbb{F}\ TASK \\
running\_tasks : cores \rightarrowtail TASK \\
executable : TASK \twoheadrightarrow cores \\
\hline
\operatorname{ran} running\_tasks \subseteq tasks \\
idles \subseteq tasks \\
\operatorname{dom} executable = tasks \\
\forall\, t : \operatorname{ran} running\_tasks \bullet running\_tasks^{\sim}(t) = executable(t) \\
\hline
\end{array}
$$

The definition for $StateData$ and $PrioData$ are similar to the FreeRTOS model. The only difference is that rather than checking only one `idle` task, we need to ensure that for all `idle` tasks, their states are `ready` or `running` and their priorities are 0. Originally, there was only one core in the system, which meant that only one physical context needed to be recorded. Here, in the new $ContextData$, a function

from $cores$ to $CONTEXT$ is used to record the physical context for each core. With these schema definitions, the schema for the base system, $Task$, can be defined as follows:

$$
\begin{array}{|l}
\hline
Task \\
\hline
TaskData \\
StateData \\
ContextData \\
PrioData \\
\hline
tasks = TASK \setminus (state^{\sim}(\!| \{nonexistent\} |\!)) \\
state^{\sim}(\!| \{running\} |\!) = \operatorname{ran} running\_tasks \\
\forall\, pt : state^{\sim}(\!| \{ready\} |\!);\ r : \operatorname{ran} running\_tasks \\
\quad | \ executable(pt) = executable(r) \bullet priority(r) \geq priority(pt) \\
\hline
\end{array}
$$

It appends three extra constraints to the system. Comparing it to the $Task$ schema in the FreeRTOS model, we can see that the meaning of these constraints are the same. However, as the number of running tasks changes from one to many, the expression has to be updated accordingly.

It is essential to provide the schema to initialise $Task$. When the system starts, (a) There are only `idle` tasks in the system; (b) The state for all tasks is $nonexistent$, except `idle` tasks, which are the $running$ tasks for their cores; (c) The physical and logical context is empty at the moment, i.e., $bare\_context$; (d) The priority is 0 for all tasks. Because each core only has an `idle` task in the initial state, the initial value for $running\_tasks$ and $executable$ should be pairs of $idles$ and $cores$. However, although it is important to know there is an `idle` task from $idles$ executing on each core from $cores$, we do not care which `idle` task maps to which core. Therefore, we leave the initial value for $running\_tasks$ and $executable$ as nondeterministic.

Similar to the previous model, the Δ schema for $Task$ has been overridden to insist that the state transfer for each task should follow the rule defined in $transition$ as well.

6.2.2 Additional Schemas

First, the schema $Reschedule$ is modified to satisfy priority-based partitioned scheduling for the multi-core environment. Specifically, all the tasks are bound to a core, which is available in the system; they will be scheduled later within the core, based on their priority. When there is a task whose priority is higher than the running task of its core, it will be scheduled as the new running task.

Reschedule

$$
\begin{array}{l}
\Delta Task \\
target? : TASK \\
tasks? : \mathbb{F}\, TASK \\
executable? : TASK \twoheadrightarrow cores \\
st? : STATE \\
pri? : TASK \rightarrow \mathbb{N} \\
\hline
target? \in tasks? \\
\mathrm{dom}\, executable? = tasks? \\
tasks' = tasks? \\
running\_tasks' = running\_tasks \\
\quad \oplus \{(executable?(target?) \mapsto target?)\} \\
executable' = executable? \\
state' = state \oplus \{(target? \mapsto running), \\
\quad (running\_tasks(executable?(target?)) \mapsto st?)\} \\
phys\_context' = phys\_context \\
\quad \oplus \{(executable?(target?) \mapsto log\_context(target?))\} \\
log\_context' = log\_context \\
\quad \oplus \{(running\_tasks(executable?(target?)) \\
\qquad \mapsto phys\_context(executable?(target?)))\} \\
priority' = pri?
\end{array}
$$

As well as the interface variables introduced in the FreeRTOS model, $target?$, $tasks?$, $st?$, $pri?$, another input variable is provided, $executable?$, which updates the executable information for tasks.

Second, there is also a frequently reused operation to search for the ready task that is bound to the same core as the given task and holds the highest priority.

findTopReady

$$
\begin{array}{l}
Task \\
target? : TASK \\
topReady! : TASK
\end{array}
$$

$$
\begin{array}{l}
target? \in tasks \\
state(topReady!) = ready \\
executable(topReady!) = executable(target?) \\
\forall\, rt : state^{\sim}(\!|\, \{ready\} \,|\!) \mid executable(rt) = executable(topReady!) \\
\quad \bullet\ priority(topReady!) \geq priority(rt)
\end{array}
$$

This schema is reasonably simple compared to the previous one. It takes the given task in the input $target?$ and returns the highest-priority ready task with $topReady!$, which is nondeterministically selected because we are not concerned about the details of any particular scheduling algorithm, only that this task holds the highest priority among the other ready tasks within the same core.

Third, due to partitioned scheduling, when creating a task it is necessary to allocate it to one specific core, which may have access to a particular resource or have the shortest distance property. If, however, the location of the task is not of interest to the user, then the system will allocate a suitable core. The algorithm used here is inspired by Best-Fit Algorithm [74] for memory allocation. The schema $findACore\_T$ is designed to be used by the task creation schemas to find a proper core for new task.

$findACore\_T$

$$
\begin{array}{l}
Task \\
newpri? : \mathbb{N} \\
executeCore? : CORE \\
executeCore : CORE \\
\hline
executeCore? \notin cores \\
executeCore \in cores \\
\exists\, tcs, cs : \mathbb{F}\, cores \mid \\
\quad tcs = \{\, pc : cores \mid newpri? > priority(running\_tasks(pc)) \,\} \\
\quad \bullet\ (tcs = \emptyset \Rightarrow cs = cores) \land (tcs \neq \emptyset \Rightarrow cs = tcs) \\
\qquad \land\ (\forall\, oc : cs \bullet executeCore \in cs \\
\qquad\quad \land\ \#(executable^{\sim}(\!|\, \{executeCore\} \,|\!)) \leq \\
\qquad\qquad \#(executable^{\sim}(\!|\, \{oc\} \,|\!)))
\end{array}
$$

The key aim of this algorithm is to find the best core for the task, so that it can be executed as soon as possible. It also attempts to minimise the maximum loads of all the cores, thus helping other tasks to meet their deadlines. Specifically, this

schema takes two input variables from the task creation schemas, $newpri?$ and $executeCore?$, which represent the new priority of the target task and the target core specified by the user. When $executeCore?$ is specified by the developer, this schema will have no effect. Otherwise, it compares the priority of the new task with the priority of all running tasks to find out whether it is possible to schedule the new task immediately. If it is possible, then the set tcs includes all possible cores. Otherwise, it is set to $cores$, which indicates all the cores available in the system. Subsequently, it examines which core has the minimum load and sets it to $executeCore$, which can be used later by related schemas.

6.2.3 Creating and Deleting Tasks

Similar to creating tasks for FreeRTOS, the relation between the priority of the new task and the priority of the running task, both of which are executed in the same core, splits task creation into two separate cases, normal and rescheduling. Depending on whether the associated core for the new task is specified, each of these cases can be further divided into two sub-cases. Therefore, the definition for task creation can be divided into four cases. When the user specifies the executing core, the new task $target?$ is added to the system with the new priority $newpri?$ and the bound core information recorded in related functions. If the priority of the new task is not higher than the priority of the running task, the system does not need rescheduling; the state of the new task is set to $ready$. Otherwise, $Reschedule$ is used to reschedule the system. When the executing core is not provided by the developer, the schema $findACore\_T$ is used to determine the core to which the new task will be bound. Therefore these cases, where the user does not provide the executing core, are covered by the cases where the executing core is specified.

The delete operation is simpler than the create operation, because it considers only two cases, rescheduling or not. These cases are similar to the FreeRTOS model. If the deleted task is not the running task, it removes the target task from the system and updates related functions; otherwise, it leads to rescheduling. As tasks are bound to different cores, we cannot simply get the task with the highest priority from the $ready$ state, like the rescheduling case for task deletion in the FreeRTOS model.

The schema $findTopReady$ is used to find which task is going to be rescheduled as the new running task for that core after the operation. Otherwise, it does a similar job to the FreeRTOS model: removes $target?$, which is a running task, from $tasks$, and $executable$; updates $topReady!$, which is defined by $findTopReady$, to be the running task in its core; updates the physical context of the executing core of $topReady!$ to its logical context; and finally, sets the logical context of the previous running task to $bare\_context$.

6.2.4 Migrating Task

As a multi-core system, it is also necessary to provide facilities for the user to move a task from one core to another. The schema, $Migration\_T$, is introduced for this purpose. There are four cases for migrating tasks:

1. $MigrationN\_T$ The task that is going to be migrated is a non-running task with a priority that is not greater than the priority of the running task of the target core, and therefore does not cause rescheduling, either in the original core or in the target core. All that needs to be done is to update the information about the target task in the new core for $executable$.

2. $MigrationS\_T$ The migrating task is a non-running task with priority higher than the running task of the new core. The original core does not perform rescheduling, but the migrating task causes a reschedule in the target core. The schema $Reschedule$ is used to do this.

3. $MigrationRuN\_T$ The migrating task is running in its core, but has a priority lower than or a priority equal to the running task of the target core. As the migrating task is moved out, rescheduling is requested to find a suitable ready task to fill the core. In the target core, however, the migrant is simply added to the ready list.

4. $MigrationRuS\_T$ The migrant task is the running task in its core and also has a higher priority than the running task in the target core. Both source and target cores need rescheduling (and $Reschedule$ cannot do this).

6.2.5 Other Operations

In addition to the operations described above, other operations are also for task management, such as suspending tasks, resuming tasks and changing the priority of tasks. However, their effects are isolated to a single core, the one in which the target task is executing. Their definitions are almost the same as in the FreeRTOS model. Therefore, they are not described in detail here. Their specification can be found in the supplementary material. It is worth noticing that as the definition of the running tasks is different, the related expressions are different and we need to keep the function $executable$ unchanged for these operations.

6.3 Queue Model

Queue is also defined as a communication facility for tasks in our system. As tasks which need to communicate with each other may be resident in the different cores, queues should be accessible for different cores. At the same time, the base statement definition for the queue model in the FreeRTOS model includes all the essential information about a queue. However, each queue is limited to a single core only. Therefore, the function q_ava is appended to the schema $QueueData$ to enable multiple accessibility for the new model. This function records a set of cores for each queue, which can access the queue. Similar to the functions, q_size and q_max, the domain of this function should also be the queues known by the system, i.e., $queue$. The range also has to be a set of cores known by the system, i.e., $cores$.

$$
\begin{array}{|l}
\hline
QueueData \\
\hline
queue : \mathbb{P}\, QUEUE \\
q\_max : QUEUE \pfun \mathbb{N}_1 \\
q\_size : QUEUE \pfun \mathbb{N} \\
q\_ava : QUEUE \pfun \mathbb{F}\, CORE \\
\hline
\dom q\_max = \dom q\_size \\
\dom q\_size = \dom q\_ava \\
\dom q\_ava = queue \\
\ran q\_ava \subseteq \mathbb{F}\, cores \\
\forall\, q : QUEUE \mid q \in queue \bullet q\_size(q) \leq q\_max(q) \\
\hline
\end{array}
$$

To extend operations from the task model to the queue level, the same strategy in Sect. 4.3.2 can be used. In addition, when deleting or suspending a task, it is necessary to remove related data from functions in $WaitingData$ and $QReleasingData$. It is worth noticing that although it is easy to use the variable $running\_task$ to refer to the task executing the operation in the FreeRTOS model, we have multiple running tasks in the new model. A new input variable, $self?$, has to be introduced to each schema which needs to refer to the task executing the operation. For instance, to distinguish whether a task has just been released from the waiting event, we check whether the running task belongs to the domain of releasing functions in the FreeRTOS model. Here, we need to verify whether $self?$ belongs to the releasing functions.

To create a queue, as well as the constraints and the behaviours described in Sect. 4.3.3, we also need to know the set of cores, $cset?$, which can access the queue. This information is recorded by appending the ordered pair, $(que?, cset?)$, to the function q_ava.

When deleting a queue, we first need to check whether the calling task belongs to a core which can access the queue. Because there is more than one running task in the system, it is impossible to refer to the calling task by using the running task. The input variable $self?$ is introduced to indicate the calling task. Then, with the function $executable$, its bound core can be identified. If this core is one of the cores which can access the queue, deleting can be performed. The information related to the queue should be removed from $queue$, q_max, q_size and q_ava.

For sending and receiving items to and from a queue, we also use the variable $self?$ to identify the calling task, and the schema $findTopReady$ is used to find the correct highest-priority ready task when rescheduling is required. The behaviours of these operations are the same as the FreeRTOS model (see Sect. 4.3.4).

Because a queue can only be accessed by a particular set of cores in the system, if a task which is using the queue migrates to another core which is not authorised to access the queue, the set of available cores for the queue has to be

updated to include the core, or the task loses its ability to access the queue. The last operation for the queue model is introduced to handle this, which is called $ChangeQueueLevel\_TQ$.

$$
\begin{array}{|l}
\hline
ChangeQueueLevel\_TQ \\
\Delta TaskQueue \\
que? : QUEUE \\
self? : TASK \\
cset? : \mathbb{F}\ cores \\
\hline
self? \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv \\
que? \in queue \\
state(self?) = running \\
executable(self?) \in q\_ava(que?) \\
cset? \neq q\_ava(que?) \\
cset? \neq \emptyset \\
\forall\, t : wait\_rcv^{\sim}(\!|\ \{que?\}\ |\!) \cup wait\_snd^{\sim}(\!|\ \{que?\}\ |\!) \\
\quad \cup release\_rcv^{\sim}(\!|\ \{que?\}\ |\!) \cup release\_snd^{\sim}(\!|\ \{que?\}\ |\!) \\
\quad\quad \bullet\ executable(t) \in cset? \\
\Xi Task \\
queue' = queue \\
q\_max' = q\_max \\
q\_size' = q\_size \\
q\_ava' = q\_ava \oplus \{(que? \mapsto cset?)\} \\
\Xi WaitingData \\
\Xi QReleasingData \\
\hline
\end{array}
$$

As we can see from the specification, the behaviour of the operation is really simple. It updates the value of $q\_ava(que?)$ and keeps everything else unchanged. However, it can be performed only when the new set of cores is not empty and not equal to the original set. In addition, it also has to include all the cores in which there are some tasks using the queue. Otherwise some of these tasks may lose access to the queue.

6.4 Time and Mutex Model

The semaphores and mutexes in this multi-core model are also defined as special cases of queues. The properties of time facilities and semaphores and mutexes are the same as the FreeRTOS model. The operation schema of the lower level model can be extended to the higher level by the same strategy. Moreover, for the

operation schemas for the time and the mutex model except $TimeSlicing\_TQT$, schemas for mutex tasking and mutex given, they focus on the behaviour of one task in one core, so their definitions are close to the FreeRTOS model. Due to length limitations, they are not repeated here (see Sect. 4.4 & 4.5 and Page. 205 and 225 for the details).

6.4.1 Time Slicing

The most interesting schema for this model is $TimeSlicing\_TQT$. When time-slicing happens, the running tasks need to be replaced in all the cores in which there are some ready tasks that have the same priority as their running task. To achieve this, we first define a set of ready tasks with the same priority as their running task, called $topReadys!$. With this set and the inverse function of $executable$, we can find all the cores which need to be rescheduled. Therefore, the following schema is given.

$TimeSlicing\_TQT$

$\Delta TaskQueueTime$
$topReadys! : \mathbb{F}\ TASK$

$\#topReadys! \leq \#cores$
$\forall\, t : topReadys! \bullet state(t) = ready$
$\quad \land\ priority(t) = priority(running\_tasks(executable(t)))$
$\forall\, t1, t2 : topReadys! \mid executable(t1) = executable(t2) \bullet t1 = t2$
$\forall\, c : cores \mid (\forall\, t : topReadys! \bullet executable(t) \neq c)$
$\quad \bullet\ (\forall\, t : executable^{\sim}(\!|\ \{c\}\ |\!) \mid state(t) = ready$
$\qquad \bullet\ priority(t) < priority(running\_tasks(c)))$
$topReadys! \neq \emptyset$
$\forall\, t : \mathrm{dom}\ time \bullet time\_slice \leq time(t)$
$tasks' = tasks$
$executable(\!|\ topReadys!\ |\!) \lhd running\_tasks'$
$\quad = executable(\!|\ topReadys!\ |\!) \lhd running\_tasks$
$executable' = executable$
$(running\_tasks(\!|\ executable(\!|\ topReadys!\ |\!)\ |\!) \cup topReadys!) \lhd state'$
$\quad = (running\_tasks(\!|\ executable(\!|\ topReadys!\ |\!)\ |\!)$
$\qquad \cup topReadys!) \lhd state$
$executable(\!|\ topReadys!\ |\!) \lhd phys\_context'$
$\quad = executable(\!|\ topReadys!\ |\!) \lhd phys\_context$
$running\_tasks(\!|\ executable(\!|\ topReadys!\ |\!)\ |\!) \lhd log\_context'$
$\quad = running\_tasks(\!|\ executable(\!|\ topReadys!\ |\!)\ |\!) \lhd log\_context$
$priority' = priority$
$\forall\, trt : topReadys!$

$$
\begin{array}{l}
\quad \bullet\ running\_tasks'(executable(trt)) = trt \\
\qquad \land\ state'(trt) = running \\
\qquad \land\ state'(running\_tasks(executable(trt))) = ready \\
\qquad \land\ phys\_context'(executable(trt)) = log\_context(trt) \\
\qquad \land\ log\_context'(running\_tasks(executable(trt))) \\
\qquad\quad = phys\_context(executable(trt)) \\
\Xi Queue \\
clock' = clock \\
delayed\_task' = delayed\_task \\
time' = time \\
time\_slice' = time\_slice + slice\_delay
\end{array}
$$

For this operation, the post condition for $running\_tasks$ has to be discussed in two parts: (a) for the cores which do not need to be rescheduled, the value has to be equal to its original value; (b) on the other hand, the value has to be updated with tasks in $topReadys!$ according to its executable core. Similarly, the post conditions for variables $state$, $phys\_context$ and $log\_context$ also need to be considered in two parts. Moreover, the value of $time\_slice$ has to be increased and the rest of the variables should be the same as before.

6.4.2 Taking Mutexes

As mentioned in Sect. 4.5.4, for a single core system, there are six cases for taking a mutex. When this operation migrates to a multi-core platform, there are two extra cases. First, if the task which is executing is taking the mutex and holds the mutex already, the specification simply increases the value of $mutex\_recursive$ for the mutex. Then, if the mutex is available, there are two cases for taking a mutex depending on whether the calling task has held a mutex already. Similar to the FreeRTOS model, the schema $QueueReceiveN\_TQT$ is used to simplify the specification. When the mutex is not available and the priority of the calling task is not greater than the priority of the mutex holder, the calling task will be blocked by the operation and no priority inheritance requested, $QueueReceiveE\_TQT$ is used for this case. In addition, once the priority of the calling task is higher than the mutex holder, the mutex holder needs to inherit the priority of the calling task. As a multi-core system, the mutex holder and the calling task may be executing in different cores. If they are executing in the same core, the operations are close

to the FreeRTOS model. The mutex holder or the top priority ready task replaces the calling task, which is blocked by the waiting event. However, when the mutex holder belongs to a different core of the calling task, there are two additional cases. When the inherited priority of the mutex holder is lower than or equal to its running task or the mutex holder is not in the ready state, for that core, no rescheduling is requested. We just need to reschedule the top priority ready task of the calling task as the new running task. On the other hand, once the mutex holder is in the ready state and the inherited priority is higher than its running task, rescheduling needs to be performed in both cores. For these two schemas, $Reschedule$ cannot be used for rescheduling. This is because of the need to update the state of three and four tasks respectively. $Reschedule$ cannot handle this.

6.4.3 Giving Mutexes

Like the FreeRTOS model, the mutex giving operation is the most complex of the whole model. The new version of $basePriorityMan$ is also introduced to help manage the base priority of the calling task. The first case for this operation is also for a recursively returning mutex. It decreases the $mutex\_recursive$ of the mutex and keeps everything else unchanged if the mutex holder takes the mutex several times. When there is no task waiting to take the mutex, we have three cases: (a) The mutex holder, i.e., the calling task, did not inherit priority from another task. It just uses the schema $QueueSendN\_TQT$ to return the mutex, removes the mutex holder from the mutex, sets the $mutex\_recursive$ of the mutex to 0, uses the schema $basePriorityMan$ to manage the base priority of the calling task and keeps the rest of the variables unchanged; (b) When the priority of the mutex holder is inherited from another task, it needs to reset its priority to the original. Further, if its original priority is still the highest of the ready tasks in the same core, it can keep executing. As the priority of the mutex holder has to be updated, the schema $QueueSendN\_TQT$ cannot be used. The behaviour has to be defined from scratch. It first updates the priority of the mutex holder, sets the size of the mutex to 1 make the mutex available for other tasks and also updates $mutex\_holder$, $mutex\_recursive$, etc. as in the previous case; (c) Finally, when the

original priority of the mutex holder is not the highest; rescheduling is requested and the schema $Reschedule$ is used. Furthermore, if there are tasks waiting to take the mutex, when the mutex holder returns the mutex, the highest priority waiting task is woken up. Meanwhile, if the mutex holder did not inherit the priority, the priority of the mutex holder does not need to be updated after the operation. What we need to do is return the mutex, and unblock the highest priority waiting task. However, depending on the priority of the woken-up task, rescheduling needs to be considered. If its priority is not higher than the running task of its core, the woken task is placed in the ready state. Otherwise, the running task is replaced in its core. The behaviour of these two cases is the same as that of the queue sending operation except variables related to mutexes. Therefore, the specification for these two schemas can be defined with the schema of $QueueSendW\_TQT$ and $QueueSendWS\_TQT$ respectively, and an update of the functions $mutex\_holder$ and $mutex\_recursive$.

Finally, different from last two cases, the mutex holder inherits priority from other tasks in the remaining cases. After the returning operation, the priority of the mutex holder has to be reset and the highest priority waiting task has to be woken up. In detail, the following conditions have to be considered to decide the behaviour of these cases:

1. Do the woken task and the calling task belong to the same core?

2. The relationship between the original priority of the mutex holder, the top priority ready task which executes in the same core as the mutex holder, the woken task and the running task in the same core as the woken task.

Table 6.1 illustrates the cases and their conditions in detail. In this table, *Running Task* refers to the running task which executes on the same core as the woken task, if the executing core for the woken up task is different from the executing core for the calling task; *Top Waiting* refers to the highest priority waiting task of the mutex; and *Top Ready* represents the highest priority ready task of the core of the calling task.

Table 6.1: Conditions for giving mutex cases (have waiting tasks and the mutex holder inherits the priority)

	Running Task	Top Waiting	Calling Task	Top Ready
1	c_2 P_H Running	c_1 - Ready	c_1 - Ready	c_1 P_H Running
2	c_2 P_H Running	c_1 P_H Running	c_1 - Ready	c_1 - Ready
3	c_2 P_H Running	c_1 - Ready	c_1 P_H Running	c_1 - Ready
4	c_2 - Ready	c_2 P_H Running	c_1 P_H Running	c_1 - Ready
5	c_2 - Ready	c_2 P_H Running	c_1 - Ready	c_1 P_H Running
6	c_2 P_H Running	c_2 - Ready	c_1 P_H Running	c_1 - Ready
7	c_2 P_H Running	c_2 - Ready	c_1 - Ready	c_1 P_H Running

From this table, firstly we can see that the effect of cases 3 & 6 are the same. They do not need to reschedule the tasks in any core. The schema $MutexGiveWInhN\_TQTM$ is defined for this case, which is the normal case for mutex returning. Secondly, cases 1 & 7 have the same effect as well. Although the relation between the executing cores for the top priority waiting task and the calling task are different in these two cases, they both request to use the top priority ready task replacing calling task as the new running task and do not reschedule for any other cores. When the woken task belongs to the core of the calling task, the priority of the top priority ready task has to be the highest compared with the original priority of the calling task and the priority of the woken task. When the operation resets the priority of the calling task, the system needs to be rescheduled. Similarly, cases 2 & 4 can also be recognised as the same case. Even through the post

states for these four tasks are different in these two cases, it can be considered as the top priority waiting task being rescheduled as the new running task in its core.

Specifically, if it belongs to the core of the calling task and its priority is the highest compared with the original priority of the calling task and the priority of the top priority ready task, it is scheduled as the new running task in its core (c_1). There is no rescheduling for other cores (c_2). On the other hand, when the woken task belongs to a different core from the calling task, and its priority is also higher than the priority of the running task in the core and the original priority of the calling task is no less than the priority of the top priority ready task in that core, the rescheduling also occurs in the core of the woken task (c_2) and no rescheduling is requested for other cores (c_1). Finally, case 5 is the last case for giving mutexes. It requests rescheduling for both cores of the calling task and the woken task. As shown in Table 6.1, the top waiting task and the calling task belong to different cores. The priority of the top waiting task and the top ready task are the highest in their cores. Rescheduling happens in both cores. As well as these differences for task related variables, the new values for the other variables are the same as the other cases of mutex giving. The specification for these schemas can be found from Page. 258.

6.5 Summary

This chapter presented the extended FreeRTOS model, which is used for a multi-core platform. The first section highlighted the differences between the original version and the multi-core version of the model. Following the structure used in Chap. 4, the model for the multi-core platform was shown, which focused on the differences between the original model and the new model. This piece of work shows that the model is easy to reuse and extend.

The next chapter will evaluate our project. Case studies will also be shown in the next chapter.

Chapter 7

EVALUATION AND CASE STUDIES

This chapter evaluates our research project. It compares the project with the objectives listed in Sect. 1.5, it also summarises the achievements of the project, followed by animations and proofs for the case studies discussed in Chap. 2. Finally, some issues about the implementation of FreeRTOS are discussed.

7.1 Project Summary

We will now show that, except objective d in Sect. 1.5 (which is included in Sect. 8.2 as a future work), our project meets the objectives (Sect. 1.5) and requirements (Sect. 3.3) proposed at the beginning of the project. During the project, we produced an abstract formal model for FreeRTOS (Obj. a). This includes 514 Z paragraphs, including basic definitions for the model, schemas for the operations, theorems for the consistency verifications and some assistant lemmas. From these paragraphs, 598 theorems were derived, all of which have been proved with Z/Eves (**Well-definedness** and **Feasibility**). In particular, around half of them (240) were proved automatically. The specified behaviour of the FreeRTOS API functions meets the requirements list in Sect. 3.3.1. The summary of the relationship between the API interfaces and the schemas in the specification is included in Appendix C. In addition, the model can also be animated by

ProZ [33] (**Animatable**). Based on our specification, the function contracts for task-related API functions were developed and the functions verified with VCC (Obj. b). However, due to time limitations, we only verified the task-related API functions and reduced the code complexity to focus on the functions modelled with the Z notation, we believe it is enough to demonstrate the possibility of directly verifying the implementation with the abstract formal specification. During the modelling and verification of FreeRTOS, we detected some issues with the FreeRTOS implementation, which are discussed in detail in Sect. 7.3 (Obj. c). Finally, we extended the model of FreeRTOS to a multi-core platform, as described in Chap. 6. The well-definedness of our multi-core model was also verified with Z/Eves, which includes the basic syntax and domain check for all the definitions. Further, the multi-core model can also be animated by ProZ (Obj. e). These extensions show that our specification for FreeRTOS is **reusable**. We also provide the source code and project file of our work in the supplementary material. This makes it easy to reproduce (**Reproducible**). Therefore, our model satisfies non-functional requirement stated in Sect. 3.3.2.

In summary, except objective d, this project meets the objectives and requirements, both functional and non-functional, proposed at the beginning of the project.

7.2 Case Studies

As discussed in Chap. 1, we have shown how our specification illustrates the execution of FreeRTOS code. Due to the different format requirements between ProZ, CZT and Z/Eves, we modified the source files to fit them. However, to animate our model with ProZ, some extra modifications were still needed, as follows:

- Remove the definition for the $findDelegate$ function, which is defined to help in proof and has no effect for animation. Most importantly, ProZ struggles with it and runs out of time;
- Remove the disabled mark for $transition$, i.e., use `\begin{zed}` to replace `\begin[disabled]{zed}`. Otherwise, ProZ cannot use this definition, as `[disabled]` is not defined in ProZ;

- Use “Init” to replace the name of the top-level initialisation schema of the model. For instance, to animate the Task model, the name of the schema $Init\_Task$, should be replaced. This is the entrance for ProZ to detect the model state and initialise the model;
- Remove the label, $\langle\langle disabled\ slice\_delay\_def \rangle\rangle$, which is not defined in ProZ;
- For the mutex model, use “$XXX?/que?$” to replace all “$que? := XXX?$” in the schema reuse, as CZT cannot recognise the second format; in addition, ProZ cannot handle the first one.
- For the mutex model, we use the schema $basePriorityMan$ (See Page. 258) to simplify the definitions for returning mutex, which is helpful for modelling. However, it dramatically increases the load for the animator. Therefore, it has to be removed and its contents have to be used to replace all calls to the schema. For example, the schema $MutexGiveNnonInh\_TQTM$ (See Page. 258) uses the schema $basePriorityMan$ to manage the base priority of the running task, which should be replaced by:

$$
\begin{aligned}
&running\_task \in \mathrm{ran}(\{mut?\} \lhd mutex\_holder) \\
&\quad \Rightarrow \Xi OriginalPriorityData \\
&running\_task \notin \mathrm{ran}(\{mut?\} \lhd mutex\_holder) \\
&\quad \Rightarrow base\_priority' = \{running\_task\} \lhd base\_priority
\end{aligned}
$$

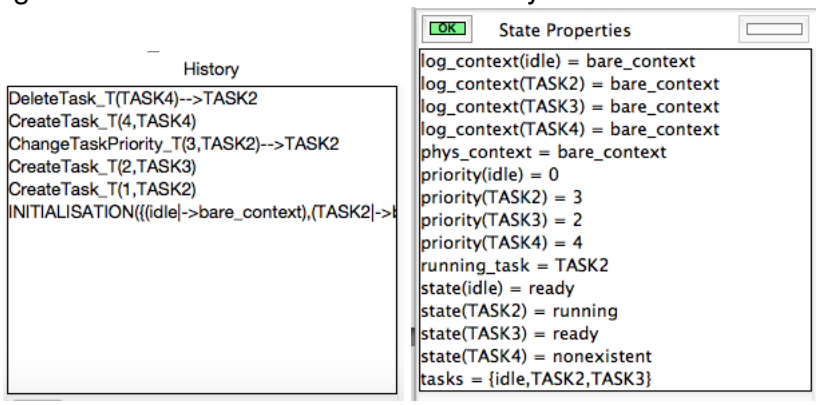
7.2.1 Case 1

The application for the first case is shown in Fig. 1.2 Page. 6. Firstly, ProZ is used to animate the model. Before animation, we set the size of our given sets to 4; the maximum integer needs to be set to 4 as well. This is because we have four tasks in the application, `idle`, `Task1`, `Task2` and `Task3`, and the maximum number used is the priority of `Task3`, namely 4. Afterwards, the `.tex` file is loaded in ProZ. Although we have three tasks in the application, these tasks execute in a single-core processor. Therefore, it is possible to predict the sequence of API function calling, which is[1]:

xTaskCreate Create `Task2` with priority of 1;

[1] As idle task is created when the animator is initialising, it is actually `Task1` for the animator. Therefore, the first task we created in the application (i.e. `Task1`) is `Task2` in ProZ.

Figure 7.1: API function execution history and result for Case 1



xTaskCreate Create `Task3` with priority of 2;

vTaskPrioritySet Change the priority of `Task2` to 3;

xTaskCreate Create `Task4` with priority of 4;

vTaskDelete Delete `Task4`;

After initialising the machine in ProZ, we call the API function in this order (see Fig. 7.1). As we analysed in Sect. 1.2.1, the expected final state of execution should be:

1. There are three tasks left in the system, `idle`, `Task2`, and `Task3`;
2. `Task2` is the running task, as it has priority of 3.
3. `Task3` is in ready state with priority 2.
4. `Task4` is unknown to the system, therefore its state is nonexistent.

As we can see from the screen shot of state properties, the result generated from our model matches our expectations.

In addition to this, we also let Z/Eves verify our result. We use a theorem, similar to Theorem: 21 (see Page. 90), to show that the behaviour of the API function matches our expectation. The API function call `xTaskCreate` is repeated three times in the application, but we only show the theorem for one of these calls. Therefore, we have the following theorems to show our model works for the application.

1. Create `Task1`. When we execute this, there is only the `idle` task in the system, which is the running task; therefore, we have to indicate this situation

to the prover. The input variables need to be introduced and we need to specify the value of $newpri?$ as 1. After this operation, we expect that the new task is created, which means it is in the set $tasks'$. It should be the running task with priority 1, which is higher than the priority of the `idle` task.

Theorem 24 (CaseStudyStep1)

$$\begin{aligned}&\forall\, Task;\ target? : TASK;\ newpri? : \mathbb{N}\\ &\quad \mid tasks = \{idle\} \land running\_task = idle\\ &\qquad \land\ newpri? = 1 \land CreateTask\_T\\ &\quad \bullet\ target? \in tasks' \land state'(target?) = running\\ &\qquad \land\ priority'(target?) = 1\end{aligned}$$

To prove this, we know that the system needs to be scheduled. Therefore, we try to eliminate the non-schedule part of the specification of $CreateTask\_T$. The key condition to distinguish these two cases is whether the priority of the new task is greater than the running task. Thus, we expand the necessary schemas of the proof goal and then let the prover discharge the proof goal automatically by the $prove\_by\_reduce;$ command.

proof $[CaseStudyStep1]$
$with\ disabled\ (CreateTaskS\_T,\ StateData,$
$TaskData,\ ContextData)\ reduce;$
$prove\ by\ reduce;$

■

2. Change the priority of `Task2` to 3. Similarly, we need to inform the prover about the pre-state of the system. The key element of the expected result of this API function call is that `Task2` is scheduled as running task with priority of 3. By eliminating the unrelated case of $ChangeTaskPriority\_T$, like the previous case for $CreateTask\_T$, it is easy to prove this theorem.

Theorem 25 (CaseStudyStep3)

$$\begin{aligned}&\forall\, Task;\ Task2, Task3, target? : TASK;\ newpri? : \mathbb{N}\\ &\quad \mid tasks = \{idle, Task2, Task3\}\\ &\qquad \land\ priority(Task2) = 1 \land priority(Task3) = 2\\ &\qquad \land\ state(Task2) = ready \land running\_task = Task3\end{aligned}$$

$$\begin{aligned}
&\wedge\ target? = Task2 \wedge newpri? = 3 \\
&\wedge\ ChangeTaskPriority_T \\
&\bullet\ priority'(Task2) = 3 \wedge running_task' = Task2
\end{aligned}$$

3. Finally, we verify the properties related to the last step of the API function call, delete `Task4`. Following the strategy introduced before, the theorem and proof can be obtained. The only difficulty in proving this theorem is in the nondeterministic definition for $topReady!$. In order to solve this, it is necessary to inform Z/Eves that (a) the possible value of $topReady!$ is one of the elements of $tasks$; and (b) the priority of $topReady!$ is the greatest amongst all $ready$ tasks, i.e., the priority of $topReady!$ has to be greater than or equal to the priority of `Task2`.

   Theorem 26 (CaseStudyStep5)

$$\begin{aligned}
&\forall\ Task;\ Task2, Task3, Task4, target? : TASK;\ newpri? : \mathbb{N} \\
&\quad \mid tasks = \{idle, Task2, Task3, Task4\} \\
&\qquad \wedge\ priority(Task2) = 3 \wedge priority(Task3) = 2 \\
&\qquad \wedge\ priority(Task4) = 4 \wedge state(Task2) = ready \\
&\qquad \wedge\ state(Task3) = ready \wedge state(Task4) = running \\
&\qquad \wedge\ target? = Task4 \wedge DeleteTask_T \\
&\quad \bullet\ state'(Task4) = nonexistent \wedge running_task' = Task2
\end{aligned}$$

7.2.2 Case 2

In Fig. 1.3, we provided example code demonstrating communication and synchronisation related API functions (see Page. 10). As described, it is also animated and verified with ProZ and Z/Eves, respectively. Similar to the previous case, we first animate it with the ProZ animator. In this case, we have two tasks in the system, together with the `idle` task. Therefore, it is necessary to set the size of the given set to 3 for this case. Similarly, 3 would be sufficient for the maximum integer, as the highest priority in the system is 3, the priority for `Task3`. After loading the model to the animator, the following sequence of API functions is called.

xSemaphoreCreateMutex Create a mutex `xMutex`;

xTaskCreate Create `Task2` with priority of 2;

xTaskCreate Create `Task3` with priority of 3;

vTaskDelay Delay `Task3` for 1 time units;

xSemaphoreTask `Task2` attempts to take the mutex `xMutex`;

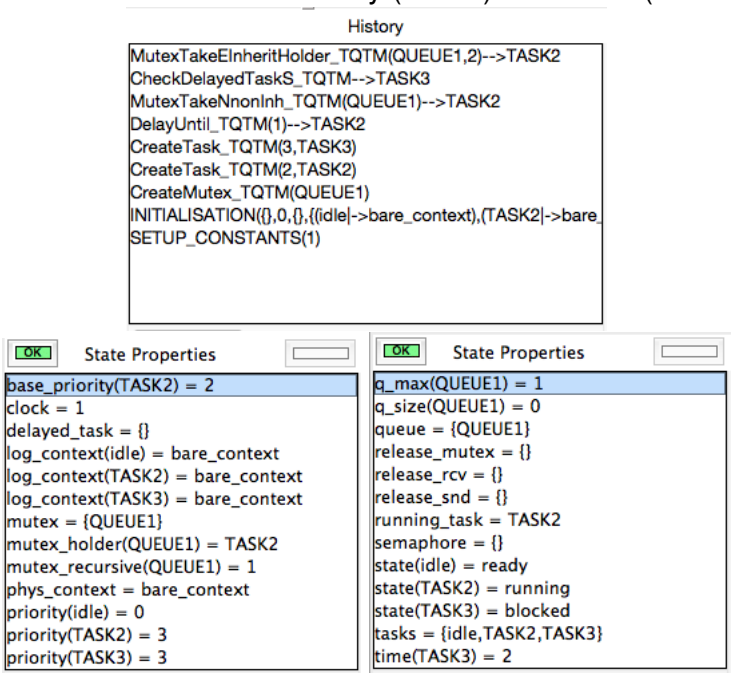
xSemaphoreTask `Task3` attempts to take the mutex `xMutex`.

After initialising the machine in ProZ, the sequence of API functions are called as shown in Fig. 7.2(Above). It should be noted that when we invoke the schema $DelayUntil\_TQTM$, we selected 1 time unit instead of 10 which was originally defined on Page. 10, because the maximum number of the integer is 3. If we increase it to 10 the load of the animator would be dramatically increased. At the same time, the purpose of this function call is to block the high priority task. This replacement has no effect on the result. Moreover, the schema $CheckDelayedTaskS\_TQTM$ is called after `Task2` has taken the mutex to release `Task3` from the blocked state. In addition, we use the sub operation schemas for mutex take instead of $MutexTake\_TQTM$. The key reason is that due to the complexity of $MutexTake\_TQTM$, if we use it directly, it considerably increases the load of the animator. The animator will therefore take a long time to calculate and may fail to respond at all.

The main behaviour of this piece of code is that two tasks compete for a single mutex. The lower priority task holds the mutex and blocks the higher priority task. As the higher priority task is blocked by the mutex taking operation, priority inheritance happens. The lower priority task inherits the priority of the higher priority task. Therefore, the final priority of `Task2` should be 3 and it should be in the running state, while `Task3` is in the blocked state with its own priority. As shown on the bottom side of Fig. 7.2-Bottom, the actual animation result matches our expectation.

Because we have shown the verification of the theorem of `xTaskCreate` in the previous section, it is not repeated here. The following theorems show two cases of `xSemaphoreTake`. The first one illustrates the case when `Task2` attempts to take the mutex. As the mutex is free initially, it successfully takes the mutex with nor-

Figure 7.2: API function execution history (above) and result (bottom) for Case 2



mal case in the specification. Meanwhile, the second represents the case when `Task3` attempts to take `xMutex`, which matches the case when the mutex is unavailable, priority inheritance is requested and the mutex holder is scheduled as the new running task. Both of them can be easily proved by the Z/Eves command "*prove by reduce*".

Theorem 27 (caseStudyTask1Take)

$$
\begin{aligned}
&\forall\, TaskQueueTimeMutex;\ mut? : QUEUE;\ topReady! : TASK \\
&\quad \mid tasks = \{idle, Task2, Task3\} \land queue = \{QUEUE1\} \\
&\qquad \land priority(Task2) = 2 \land priority(Task3) = 3 \\
&\qquad \land state(Task3) = blocked \land running\_task = Task2 \\
&\qquad \land mut? = QUEUE1 \land QUEUE1 \notin \operatorname{dom} mutex\_holder \\
&\qquad \land QUEUE1 \in mutex \land release\_rcv = \emptyset \\
&\qquad \land release\_snd = \emptyset \land base\_priority = \emptyset \\
&\qquad \land MutexTakeNnonInh\_TQTM \\
&\quad \bullet\ mutex\_holder'(QUEUE1) = Task2 \\
&\qquad \land priority'(Task2) = 2 \land priority'(Task3) = 3 \\
&\qquad \land running\_task' = Task2
\end{aligned}
$$

Theorem 28 (caseStudyTask2Take)

$$
\begin{aligned}
&\forall\, TaskQueueTimeMutex;\ mut? : QUEUE;\ topReady! : TASK \\
&\quad \mid tasks = \{idle, Task2, Task3\} \land queue = \{QUEUE1\}
\end{aligned}
$$

$$
\begin{aligned}
&\wedge\ priority(Task2) = 2 \wedge priority(Task3) = 3 \\
&\wedge\ state(Task2) = ready \wedge running\_task = Task3 \\
&\wedge\ mut? = QUEUE1 \wedge QUEUE1 \in mutex \\
&\wedge\ QUEUE1 \in \mathrm{dom}\ mutex\_holder \\
&\wedge\ mutex\_holder(QUEUE1) = Task2 \\
&\wedge\ release\_rcv = \emptyset \wedge release\_snd = \emptyset \\
&\wedge\ base\_priority = \emptyset \wedge clock = 1 \wedge n? = 2 \\
&\wedge\ MutexTakeEInheritHolder\_TQTM \\
\bullet\ &mutex\_holder'(QUEUE1) = Task2 \wedge state'(Task3) = blocked \\
&\wedge\ priority'(Task2) = 3 \wedge priority'(Task3) = 3 \\
&\wedge\ running\_task' = Task2
\end{aligned}
$$

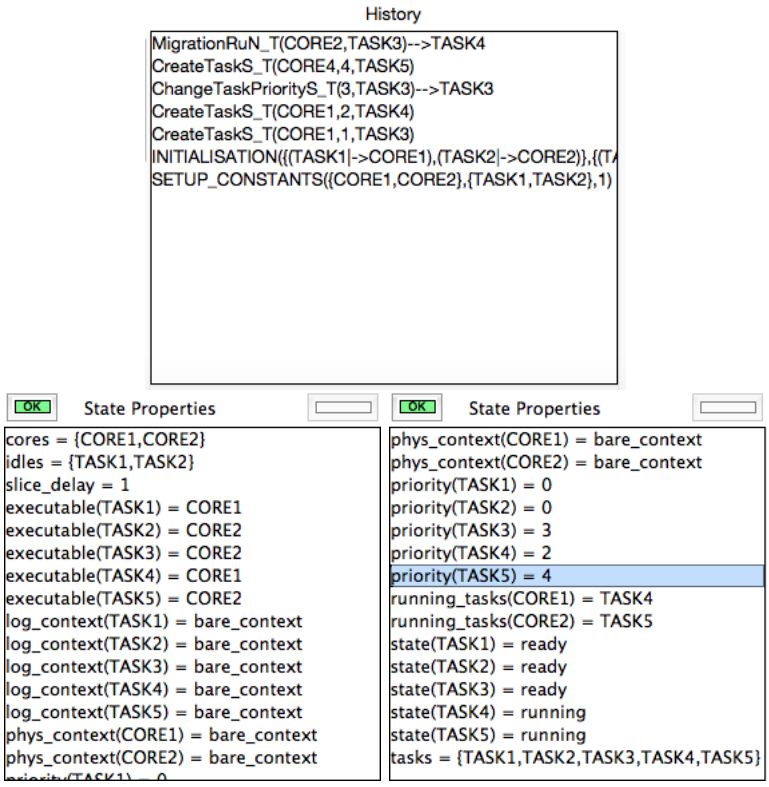
7.2.3 Case 3

Our extension model for the multi-core platform can also be animated with ProZ. Similarly, we formulate some sequences of operation calls that illustrate the behaviour of the model. Consider the following sequence, assuming there are two cores available to the system.

1. Initially, there are two tasks, `Task3` and `Task4`, created on `Core1` with priority of 1 and 2, respectively.
2. Change the priority of `Task3` to 3.
3. Create `Task5` with priority of 4.
4. Move `Task3` to `Core2`.

When the system is initialised, `Task4` occupies `Core1`, as it has the highest priority within `Core1`. Meanwhile, on `Core2`, the `idle` task is executing. When the priority of `Task3` is changed to 3, it preempts `Task4` and can execute. Thereupon, `Task5` is created with a priority of 4. As it is not specified on which core the new task is created, the system selects one to accommodate the new task, based on the algorithm described in Sect. 6.2.3. In this case, `Task5` will be created on `Core2`. Finally, `Task3` is moved to `Core2`. Because `Task3` is the current running task on `Core1`, moving it to `Core2` causes `Task4` to be scheduled as the next running task. Therefore, the final state of the system should be as follows: (a) there are five tasks in the system; (b) `Task4` and `Task5` are executing on `Core1` and `Core2`, respectively; (c) `Task3` is ready in `Core2`. To animate this

Figure 7.3: API function execution history (above) and result (bottom) for Case 3



process, we need at least five individual tasks and two cores available from the animator with a maximum natural number of 4. We initialise the system with the `setup_constants` and `initialisation` commands. For the `setup_constants` command, we need sets $\{CORE1, CORE2\}\&\{TASK1, TASK2\}$ as the parameters. Next, the `initialisation` command is used to initialise the system with maplets $TASK1 \mapsto CORE1$, $TASK2 \mapsto CORE2$ and so on, as parameters. This initialises the system with some definitions, such as, $TASK1$ and $TASK2$ as the `idle` task for $CORE1$ and $CORE2$, respectively. At this moment, we can animate the system following the sequence described above (see Fig. 7.3-Above). After these operations, we find the system status has been changed to Fig. 7.3(Bottom), which matches our expectation.

Again, we can verify the API functions with the Z/Eves theorem prover. For instance, consider the scenario where we initialise the state as shown above and then call $CreateTask\_T$ to create `Task3` on `Core1` with priority 1.

Theorem 29 (createTaskOverTwoCores)

$$\begin{aligned}
&\forall\, Task;\ target? : TASK;\ newpri? : \mathbb{N};\ c? : CORE \\
&\mid cores = \{c1, c2\} \land running_tasks = \{(c1 \mapsto i1), (c2 \mapsto i2)\} \\
&\quad \land\, newpri? = 1 \land tasks = \{i1, i2\} \land c? = c1 \\
&\quad \land\, executable = \{(i1 \mapsto c1), (i2 \mapsto c2)\} \land CreateTask_T \\
&\bullet\ target? \in tasks' \land state'(target?) = running \\
&\quad \land\, priority'(target?) = 1 \land executable'(target?) = c1
\end{aligned}$$

, where $i1$, $i2$ represent idle tasks for two cores (i.e. $c1$ and $c2$), respectively.

7.3 Issues of FreeRTOS

During the modelling and verifying process of our project, we detected some issues with FreeRTOS. These issues were revealed at different stages of the project. For instance, when we analysed the implementation to complete the requirements for modelling, we found an issue with changing the priority of a mutex holder (see, item 5). When we proved the precondition theorems, we found that deleting the `idle` task violated the system constraints (see, item 1) and so on. Due to the limitation of our VCC model, only item 1 can be detected during VCC verification. But it shows the possibility of verifying the implementation with the VCC and the abstract specification.

1. As shown in the precondition for `vTaskDelete` (Sect. 5.4), there is one predicate, which states the target task does not equal the `idle` task. If this predicate is removed, the verification would fail. The reason is that the operation may delete the `idle` task, which breaks the constraints of the system. Generally, the handler of the `idle` task is hidden from the user. However, it is possible to obtain it from `vApplicationIdleHook`, which is provided by FreeRTOS and used to define extra behaviours of the `idle` task. Therefore, we need an extra precondition to ensure that the `idle` task cannot be deleted by the operation.

2. Similar to the previous issue, the delete API function does not check if the target is the holder of a mutex. Because our VCC model does not include the

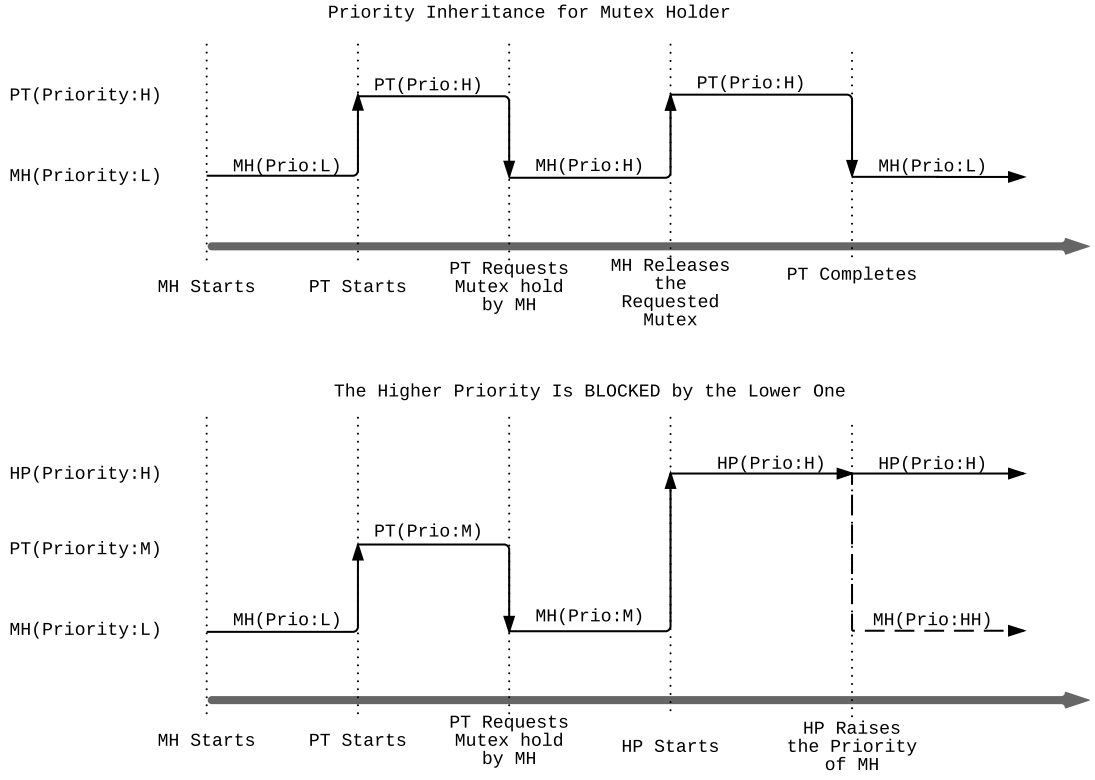
context related to mutex, this issue has no effect on our verification. However, if a mutex holder is deleted without returning its mutex, then the resource is locked permanently.

3. According to [19], if `vTaskSuspendAll` is called, `xTaskResumeAll` has to be called as many times as the suspend operation to resume the scheduler. FreeRTOS uses an unsigned counter, `uxSchedulerSuspended`, to record this. However, the API function `vTaskSuspendAll` does not check the overflow of the counter. In other words, when it overflows, only one call to `xTaskResumeAll` can resume the scheduler from millions of calls of `vTaskSuspendAll`. Although calling `vTaskSuspendAll` millions of times might not happen in real life, this could still be a weak point in the system.

4. Similar to the issues related to `vTaskDelete`, the API function `vQueueDelete` also does not verify that the target queue is not in use. If a task is blocked by waiting to send/receive an event to/from the queue, when it is woken up before the expiration time, it attempts to continue its operation. At this moment, the target queue could be an invalid pointer, if the queue has been deleted. Furthermore, in FreeRTOS, the API function `vSemaphoreDelete` is used to delete a mutex, which actually uses `vQueueDelete` directly to perform the deleting operation. It is then possible to delete a mutex, which is held by a task. This can even be performed by a task which is not the holder of the mutex. When the holder returns the mutex or recursively takes it, the handler of the mutex can be an invalid pointer again.

5. Finally, we reveal an issue, which is not about the implementation, but related to the design. When the running task fails to take a mutex of which the holder has a lower priority, then the higher priority task is blocked and a lower priority task can be executed prior to a high priority task. This is called priority inversion. FreeRTOS adopts priority inheritance to solve this. To implement this, a base priority is introduced for each task. This records the original priority of each task. Afterwards, when a user calls the API function

Figure 7.4: Scenario for priority inversion issue



to change the priority of the mutex holder, which has inherited priority from another task, FreeRTOS only updates its base priority. Generally, this works fine. However, if the new priority of the mutex holder is even higher than its inherited priority, this can cause priority inversion again.

Consider the following scenario (see Fig.7.4). The upper chart shows the normal case for priority inheritance. When a higher priority task requests a mutex held by a lower priority task, the lower priority task inherits the priority of the higher priority task. When it is scheduled as running, it finishes the job, releases the mutex and recovers its own priority. Then the higher priority task can take the mutex and continue its task. This works fine in this case. However, the problem happens at the end of the scenario shown in the lower chart. There are three tasks in this scenario, `MH`, `PT`, `HP`, which represent the mutex holder, the preemptive task, and the higher priority task respectively. Firstly, `MH` starts running and takes the mutex. When the preemptive task, which has a higher priority than the mutex holder, joins the system, it pre-

empts the system and starts to execute. It then attempts to take the mutex and is blocked by the failure to take the unavailable mutex. MH resumes its execution and inherits priority from the PT. Before it finishes its job and releases the mutex, HP preempts the system and updates the priority of MH to a even higher priority than HP. Theoretically, the mutex holder has the highest priority in the system at the moment, since it should be running. However, because FreeRTOS only updates the base priority of the mutex holder, which is not used for scheduling, HP would continues its execution.

We have produced sample code for each of these cases, which are included in the supplementary material. They can be directly used with the FreeRTOS v7.3.0 simulator.

7.4 Summary

This chapter has evaluated our project in terms of achievement and the case study. It first described the achievement of this project against the objectives, which were stated in Sect. 1.5, and requirements, which were abstracted in Chap. 3. Then, it illustrated the animation and theorem proof for case studies in ProZ and Z/Eves respectively to show the correctness of the model.

It is worth noting that the theorems for the case studies are composed manually. Due to the complexity of the system, it is possible that there are mistakes in these theorems. Especially, when a mistake happens in the condition part of a theorem, the theorem is still able to be proved by Z/Eves. Because, according to the definition of implication, the false condition can imply anything. Therefore, if there is contradiction in the condition part, it is extremely difficult to locate. To avoid this kind of problem, we wrote some simple theorems to check the condition for these theorems, which simply state the condition implies to false. If there is a contradiction in the condition, which makes the condition false, the theorem can be proved easily by Z/Eves. Otherwise, the condition is correct.

Chapter 8

CONCLUSION AND FUTURE WORK

This chapter concludes the project, including experience gained during the project. It discusses possible future work for the project and reports on some attempts related to future work.

8.1 Conclusion

We have produced the first complete abstract specification of FreeRTOS. The model can be animated by the ProZ tool to show how FreeRTOS works. We have shown that the model is internally consistent by discharging all the verification conditions for well-definedness of the specification and by calculating the exact preconditions for the successful operation of each part of the FreeRTOS API function. Experiments were performed based on this model.

Firstly, we translated the base state of the model, and pre and postconditions of task-related schemas into the virtual model structure and function contracts of VCC, respectively, to build an annotated version of the FreeRTOS implementation, which was then verified by VCC.

However, due to time limitations, we were only able to focus on task API functions and a simplified implementation. This still shows the possibility of verifying software implementation directly with a high-level abstract specification and a code

verifier. The typical way to obtain the implementation in formal methods is refinement, which is normally complex and time consuming. Although it guarantees the correctness of the design and the implementation, it also increases the difficulty and expense of the development. However, not every piece of software requires this high level of correctness, and for such pieces of software, we expect the way we have demonstrated to be an easier approach to formalisation.

Secondly, we have developed an extended model for a multi-core platform. The requirements of the specification were inspired (but not limited) by the Multi-BSP model [35]. As a high-level abstract model, we described the general behaviours for each operation, which can then be refined for a specific architecture. We also validated the model with the theorem prover, Z/Eves, by performing the syntax and domain checking for all the definitions. This work demonstrates the reusability of our verified specification. It also builds the foundations for developing a verified RTOS for the multi-core platform.

During the project, there were plenty of difficulties in both modelling and verifying, for instance:

1. Some operations (e.g., sending items to a queue) may have intermediate states and it was necessary to find the correct way to describe them.

2. The Z/Eves prover provided reasonable proof automation. However, it also had some problems. For example, in some cases, the order of predicates in a schema had an effect on the result of a proof, which should not be the case.

3. We also found that although there were many similar proof goals, we needed to guide the prover to prove them repeatedly. This increased the work load dramatically. A well-designed proof structure would be helpful for simplifying the proof process.

4. We spent considerable time on VCC experiments. It was not as easy as we imagined at the beginning. For simple cases, it was very clear and easy.

However, when the hierarchical ownership tree, which is one of the most important concepts of VCC, became involved, it was easy to become confused. Ideally, VCC should notice and highlight all false assertions or predicates that lead to conflict. However, if conflicts happen in the code, VCC may consider $false$ to be $true$, which makes later proofs meaningless. Furthermore, the error model report provided by VCC was not detailed enough for the user to understand why the proof failed. This increased the difficulty of working with VCC.

We also learnt that the Z notation is sufficient for verifying the correctness of a single function, which is similar to the unit test in normal software engineering. However, for some of the system properties of the RTOS (such as time-related issues), other techniques have to be used (like CSP).

We provide all the definitions and proofs in the supplementary material, so that the entire verification can be replayed to check its authenticity. This means that crucially our experiment is repeatable.

8.2 Future Work

For future work,

1. As mentioned in Chap. 6, promotion is an alternative way to extend our specification to a multi-core platform. This would be the first project we expect to finish as future work and provides several benefits for the specification. The most important is the possibility for code reuse during development. We have finished the first part of the model, the task model, with the Z promotion approach, which is described below.

2. Currently, we translated Z specification to VCC manually without any verification about data refinement. As Z notation provides richer data types and operations than VCC, we would expect that the data types of variables in our FreeRTOS model can be refined to data types, which can be directly used

with VCC, in Z notation and verified with Z/Eves automatically. Then directly use these refined variables in VCC. This will increase the reliability of our verification during translation process. Furthermore, when the data refinement is achieved and the variables in the Z specification can be directly used in VCC, the translation process will become more straightforward. We would also expect the translation process can be performed automatically by some software.

3. We expect to verify the whole implementation of FreeRTOS with VCC and our abstract specification. This will not only provide more solid evidence to support our expectations in the previous section, but also assure the correctness of the implementation of FreeRTOS.

4. The models, both for FreeRTOS and multi-core RTOS, can be refined, level by level, to executable source code in order to obtain a fully verified implementation of RTOS. This will provide a highly assured RTOS for industrial and research use.

We are keen to encourage others to use our specifications and proofs as benchmarks for comparing other notation and tools.

8.3 Task Model with Promotion

We attempted to apply the Z promotion technique to improve our multi-core model. The model[1] for task API functions is developed as an example to illustrate how that can be achieved, and how it can also be animated with ProZ [33]. To promote the task model for FreeRTOS to a multi-core platform, we consider that there is a sub system, which is an instance of FreeRTOS, executing in each core. Therefore, we can use the task model for FreeRTOS, except the definition for the `idle` task, directly as a part of the new model. In the FreeRTOS model, we define the `idle`

[1] Although it can be very slow, this model can be animated with ProZ with some modification, which is stated in the comments of the source file. We also provide example animation in the supplementary material.

task as a global constant of $TASK$. However, for the multi-core model, we need one `idle` task for each core. We modify the definition of the `idle` task for a core as a "local" variable of the schema $TaskData$ and introduce constants, $idles$ and $cores$, globally with constraints similar to the model shown in Chap. 6.

$$
\begin{array}{|l}
\hline
TaskData \\
\hline
tasks : \mathbb{F}\ TASK \\
running\_task : TASK \\
idle : TASK \\
\hline
running\_task \in tasks \\
idle \in tasks \\
idle \in idles \\
\hline
\end{array}
$$

As the `idle` task should never change for all operations, we appended an additional constraint to the schema $\Delta Task$, which states that the `idle` task is equal *before* and *after* the operation. The rest of the task model can be used directly. The base state schema for the multi-core model can then be defined as follows:

$$
\begin{array}{|l}
\hline
Multi\_Task \\
\hline
subTask : cores \rightarrow Task \\
exeCore : TASK \twoheadrightarrow cores \\
\hline
\forall\, c1, c2 : cores \mid c1 \neq c2 \bullet \\
\quad (subTask(c1)).tasks \cap (subTask(c2)).tasks = \emptyset \\
\mathrm{dom}\ exeCore \in \mathbb{F}\ TASK \\
\mathrm{dom}\ exeCore = \bigcup \{ c : cores \bullet (subTask(c)).tasks \} \\
\hline
\end{array}
$$

Firstly, the total function, $subTask$, is used to match cores and their local FreeRTOS. In Z, a state schema (e.g., $Task$) can be used as a data type. Its characteristic binding can be obtained by the operator "θ", which binds the values of variables of an instance of the schema to the name of the variables of the schema, and this can then be used to assign the instance of the schema to a variable. Simply, it can be understood as a handler of an instance of the schema. Similar to the model described in Chap. 6, the partial surjective function, $exeCore$, is given to record the relationship between the tasks and their executable core. Three constraints are introduced to describe the properties of the system. First, we state that the $tasks$ set for each core are disjoint; then, the domain of $exeCore$ is defined as the union of

all the $tasks$ for each core; based on the previous constraints, we can easily prove that the domain of $exeCore$ has to be a finite set, because $tasks$ for each core and the set of all the cores in the system, $cores$, are finite sets. However, it is hard to prove this with Z/Eves. We chose to append another constraint to allow Z/Eves to recognise this. Initially, there were only the `idle` tasks for each core in the system. Therefore, we defined the domain of the initial state of $exeCore$ as equal to $idles$. In addition to this, the initialise schema for the task model, $Init\_Task$, was used to help us to initialise the system for the multi-core model by initialising each subsystem for each core.

$$
\begin{array}{|l}
\hline
Init \\
Multi\_Task' \\
\hline
\mathrm{dom}\ exeCore' = idles \\
\forall\, c : cores \\
\quad \bullet\ \exists\, Task' \mid Init\_Task \\
\qquad \bullet\ subTask'(c) = \theta Task' \land exeCore'((subTask'(c)).idle) = c \\
\hline
\end{array}
$$

Based on these definitions, the promotion schema was defined. This describes the link between the global and local operations. In this case, three promotion schemas are specified, because the creating, deleting, and remaining operations have different behaviours. Specifically:

1. The promotion schema for the creating operations can be defined as the schema, $PromoteC$, which takes $\Delta Task$, the input variable (i.e., $target?$) and an extra temporary variable (i.e., $executeCore$) as parameters. In particular, $\Delta Task$ is actually provided by the local operation of FreeRTOS. As a new task, the value of $target?$ should not be in the system. The first precondition is used to restrict the target task from being included in the domain of $exeCore$. At the same time, we need to identify which instance of $Task$ is executing in the core, $executeCore$. The second predicate of the preconditions is used to achieve this. Finally, for the post-state of operations, the maplet of $executeCore$ and the characteristic binding of the post state of $Task$ is updated to $subTask$ and the relation between $target?$ and $executeCore$ is added to $exeCore$.

$$
\begin{array}{|l}
\hline
PromoteC \\
\Delta Multi\_Task \\
\Delta Task \\
target? : TASK \\
executeCore : cores \\
\hline
target? \notin \mathrm{dom}\ exeCore \\
subTask(executeCore) = \theta Task \\
subTask' = subTask \oplus \{executeCore \mapsto \theta Task'\} \\
exeCore' = exeCore \oplus \{target? \mapsto executeCore\} \\
\hline
\end{array}
$$

2. Unlike the promotion operation for creating tasks, we need to remove the target task from the system. The promotion schema for deleting tasks, $PromoteD$, needs to remove $target?$ from $exeCore$, instead of adding it to $exeCore$. The other constraints should be the same, in order to identify the local $Task$ and update it with the post state of $Task$ to $subTask$.

3. For the remaining operations, which can be described with promotion, there is no effect on the function executable core, whether suspending tasks, resuming tasks or changing the priority of tasks. Similar to the previous case, the difference between the promotion schemas, $Promote$ and $PromoteC$, is the function $exeCore$, which stays the same.

Furthermore, with the promotion schemas, the sub-definitions for creating tasks can be defined as:

$$createTaskN\_MT \mathrel{\widehat=} \exists \Delta Task \bullet CreateTaskN\_T \land PromoteC$$

$$createTaskS\_MT \mathrel{\widehat=} \exists \Delta Task \bullet CreateTaskS\_T \land PromoteC$$

Like the multi-core model described in Chap. 6, the schema, $findACore\_MT$, is introduced to locate the best position for a new task if its executable core is not specified by a user. The creating operation can be defined similarly. The same strategy can be applied to specify the rest of the operations.

We also specify the behaviours of migrating a task from one core to another, which also contains four cases, as in Sect. 6.2.4. However, the definitions are slightly

more complex than those for the old multi-core model. The reason for this is that in the old model, the function $state$, $priority$, *etc.*, are global total functions, which are easy to access by the operation schema. Instead, in the promotion model, they are private for each subsystem and can only be accessed by the function $subTask$. Due to the complexity of the expressions, we introduce two assistant variables, $srcSys$ and $tarSys$, to represent the subsystems for the original core and the target core respectively. The behaviour of this operation can then be defined similarly to the one for the old model.

8.4 Summary

This chapter has summarised the project and suggested some future works after the project. It firstly described the whole process of the project, and identified the difficulties we experienced during the project. It then shows the potential future works based on the results of the project. Finally, as an example, we presented our attempt on specifying multi-core task model with promotion technique.

In general, in this project, we produced the first complete abstract specification in Z of FreeRTOS, together with proofs of consistency (well-definedness, initialisation, precondition, and a few properties). Then, the model is extended for multicore platform, with basic proofs of well-definedness (including syntax checking and domain validating). The abstract characterisation of both models is a first step towards a verified implementation of FreeRTOS on multicore. We were the first to promote FreeRTOS as a pilot project in VSI, and the work presented continues this by establishing a benchmark for others to follow. We believe that this is an important contribution to both the verification community and also the embedded systems community. We also demonstrated the possibility to verify a software system by combining the formal modelling (Z model) and code verifier (VCC), which can be an easier approach for improving the quality of the software.

Appendix A

INTRODUCTORY APPENDIX

There are twelve appendixes provided after this introductory appendix. They contain auxiliary information for the main body of the thesis. Specifically,

(a) The summary of frequently used proof commands is shown in Appendix B. It is helpful for understanding the proof script provided in supplementary material.

(b) As stated in Sect. 4.6, the mapping between API functions and the precondition for schema interface from FreeRTOS model can be found in Appendix C.

(c) The specification for our FreeRTOS model can be found in Appendix D-G. They include all the definitions, schemas and theorems (i.e. precondition theorems for operation schemas, auxiliary theorems and theorems for some system properties). Unfortunately, the proof scripts for the model cannot be included, due to the length of the script itself. However, they can be found in supplementary material and can be used directly in the theorem prover, Z/Eves.

(d) Similarly, the specification for multi-core model can also be found in Appendix H-K. And the the specification for the multi-core task model with promotion technique is shown in Appendix L.

(e) Finally, the VCC annotated source code of task related API functions (i.e. creating, deleting, suspending, resuming tasks and changing the priority of tasks) is listed in Appendix M.

Appendix B

SUMMARY OF Z/EVES PROOF COMMANDS

We summarise the proof commands used in proving of the model. For full instruction of the proof commands, please see Chap. 5 of [32].

prove The prover automatically applies sequences of proof commands. For example, $simplify$, $rewrite$, $rearrange$. Besides this, the mathematical rules included in Z/Eves' mathematical toolkit [70] are applied, if possible.

prove by reduce The prover repeatedly reduces the current proof goal. In addition to what **prove** does, the prover expands all names.

with enabled (theorem) This is a prefix that is applied to the $prove$, $prove\ by\ reduce$, or an already prefixed command. Many inefficient rules are disabled by default, and this prefix enables them for the current command. For example, $with\ enabled\ (applyOverride)\ prove$ allows the prover to use the disabled theorem $applyOverride$ within the scope of the $prove$ command.

with disabled (theorem) This is similar to the previous command, except that it disables the theorem rather than enabling it.

with normalization This is also a prefix for prove commands. It allows the prover to use “if-then-else” normal-form to represent all logical connectives [70].

instantiate This command allows the prover to instantiate quantified variables (universal in the assumptions, existential in the goal).

apply theorem As mentioned above, there are plenty of disabled rules in Z/Eves' mathematical toolkit. This command applies the specified theorem to rewrite the goal.

use theorem This command allows a specified theorem to be used to deduce additional assumptions.

extensionality A theorem included in Z/Eves Mathematical Toolkits [70], which defined as:

$$X = Y \Leftrightarrow (\forall\, x : X \bullet x \in Y) \land (\forall\, y : Y \bullet y \in X)$$

Appendix C

SUMMARY OF INTERFACE

Table C.1: API mappings & preconditions for operations

API	Operation	Precondition
xTaskCreate	$CreateTask\_T \mathrel{\widehat=} CreateTaskN\_T$	$state(target?) = nonexistent$
	$\quad \lor\ CreateTaskS\_T$	
	$CreateTaskN\_T$	$state(target?) = nonexistent$
		$newpri? \leq priority(running\_task)$
	$CreateTaskS\_T$	$state(target?) = nonexistent$
		$newpri? > priority(running\_task)$
vTaskDelete	$DeleteTask\_T \mathrel{\widehat=} DeleteTaskN\_T$	$target? \in tasks \setminus \{idle\}$
	$\quad \lor\ DeleteTaskS\_T$	$state(target?) = running \Rightarrow$
		$\quad (\exists\, topReady! : state^{\sim}(\!\mid \{ready\} \mid\!)$
		$\quad\quad \bullet\ (\forall\, t : state^{\sim}(\!\mid \{ready\} \mid\!)$
		$\quad\quad\quad \bullet\ priority(topReady!) \geq priority(t)))$
	$DeleteTaskN\_T$	$target? \in tasks \setminus \{idle\}$
		$state(target?) \in \{ready, blocked, suspended\}$
	$DeleteTaskS\_T$	$target? \in tasks \setminus \{idle\}$
		$state(target?) \in \{running\}$
		$\exists\, topReady! : state^{\sim}(\!\mid \{ready\} \mid\!)$
		$\quad \bullet\ (\forall\, t : state^{\sim}(\!\mid \{ready\} \mid\!)$
		$\quad\quad \bullet\ priority(topReady!) \geq priority(t))$
-	$ExecuteRunningTask\_T$	$\exists\, phys\_context' : CONTEXT$
		$\quad \bullet\ phys\_context' \neq phys\_context$

Table C.2: API mappings & preconditions for operations(continue)

vTaskSuspend	$SuspendTask\_T \mathrel{\widehat=} SuspendTaskN\_T$ $\lor\ SuspendTaskS\_T$ $\lor\ SuspendTaskO\_T$	$target? \in tasks \setminus \{idle\}$ $state(target?) = running \Rightarrow$ $(\exists\, topReady! : state^{\sim}(\!\mid \{ready\} \mid\!)$ $\bullet\ (\forall\, t : state^{\sim}(\!\mid \{ready\} \mid\!)$ $\bullet\ priority(topReady!) \geq priority(t)))$
	$SuspendTaskN\_T$	$target? \in tasks \setminus \{idle\}$ $state(target?) \in \{ready, blocked\}$
	$SuspendTaskS\_T$	$target? \in tasks \setminus \{idle\}$ $state(target?) \in \{running\}$ $\exists\, topReady! : state^{\sim}(\!\mid \{ready\} \mid\!)$ $\bullet\ (\forall\, t : state^{\sim}(\!\mid \{ready\} \mid\!)$ $\bullet\ priority(topReady!) \geq priority(t))$
	$SuspendTaskO\_T$	$state(target?) = suspended$
vTaskResume	$ResumeTask\_T \mathrel{\widehat=} ResumeTaskN\_T$ $\lor\ ResumeTaskS\_T$	$state(target?) = suspended$
	$ResumeTaskN\_T$	$state(target?) = suspended$ $priority(target?) \leq priority(running\_task)$
	$ResumeTaskS\_T$	$state(target?) = suspended$ $priority(target?) > priority(running\_task)$

Table C.3: API mappings & preconditions for operations(continue)

vTaskPrioritySet	$ChangeTaskPriority\_T$	$state(target?) \neq nonexistent$
	$\widehat{=}\ ChangeTaskPriorityN\_T$	$target? = idle \Rightarrow newpri? = 0$
	$\vee\ ChangeTaskPriorityS\_T$	$\exists\, topReady! : state^{\sim}(\!\mid \{ready\} \mid\!)$
	$\vee\ ChangeTaskPriorityD\_T$	$\bullet\ (state(target?) \in \{running\}$
		$\wedge\ \neg\ (\forall\, rtsk : TASK \mid state(rtsk) = ready$
		$\bullet\ newpri? \geq priority(rtsk)))$
		$\Rightarrow newpri? < priority(topReady!)$
		$\wedge\ (\forall\, t : state^{\sim}(\!\mid \{ready\} \mid\!)$
		$\bullet\ priority(topReady!) \geq priority(t))$
	$ChangeTaskPriorityN\_T$	$state(target?) = ready \Rightarrow$
		$newpri \leq priority(running\_task)$
		$state(target?) = running$
		$\Rightarrow (\forall\, t : state^{\sim}(\!\mid \{ready\} \mid\!) \bullet newpri? \geq priority(t))$
		$state(target?) \neq nonexistent$
		$target? = idle \Rightarrow newpri? = 0$
	$ChangeTaskPriorityS\_T$	$state(target?) = ready$
		$newpri? > priority(running\_task)$
		$target? = idle \Rightarrow newpri? = 0$
	$ChangeTaskPriorityD\_T$	$state(target?) \in \{running\}$
		$target? = idle \Rightarrow newpri? = 0$
		$\exists\, topReady! : state^{\sim}(\!\mid \{ready\} \mid\!)$
		$\bullet\ newpri < priority(topReady!)$
		$\wedge\ (\forall\, t : state^{\sim}(\!\mid \{ready\} \mid\!)$
		$\bullet\ priority(topReady!) \geq priority(t))$

Table C.4: API mappings & preconditions for operations(continue)

xQueueCreate	$CreateQueue\_TQ$	$que? \notin queue$ $size? > 0$
xQueueDelete	$DeleteQueue\_TQ$	$que? \in queue$ $que? \notin \text{ran}\ wait\_snd \cup \text{ran}\ wait\_rcv$ $que? \notin \text{ran}\ release\_snd \cup \text{ran}\ release\_rcv$
xQueueSend	$QueueSend\_TQ$ $\widehat{=}\ QueueSendN\_TQ$ $\lor\ QueueSendW\_TQ$ $\lor\ QueueSendWS\_TQ$ $\lor\ QueueSendF\_TQ$	$que? \in queue$ $q\_size(que?) = q\_max(que?) \Rightarrow running\_task \neq idle$
	$QueueSendN\_TQ$	$que? \in queue$ $q\_size(que?) < q\_max(que?)$ $que? \notin \text{ran}\ wait\_rcv$
	$QueueSendW\_TQ$	$que? \in queue$ $q\_size(que?) < q\_max(que?)$ $\forall\, wr : wait\_rcv^{\sim}(\!\vert\ \{que?\}\ \vert\!) \bullet priority(running\_task) \geq priority(wr)$
	$QueueSendWS\_TQ$	$que? \in queue$ $q\_size(que?) < q\_max(que?)$ $\exists\, topReady! : wait\_rcv^{\sim}(\!\vert\ \{que?\}\ \vert\!)$ $\quad \vert\ (\forall\, wr : wait\_rcv^{\sim}(\!\vert\ \{que?\}\ \vert\!) \bullet priority(topReady!) \geq priority(wr))$ $\quad \bullet\ priority(topReady!) > priority(running\_task)$
	$QueueSendF\_TQ$	$que? \in queue$ $q\_size(que?) = q\_max(que?)$ $running\_task \neq idle$

Table C.5: API mappings & preconditions for operations(continue)

xQueueReceive	$QueueReceive\_TQ$	$que? \in queue$
	$\widehat{=}\ QueueReceiveN\_TQ$	$q\_size(que?) = 0 \Rightarrow running\_task \neq idle$
	$\lor\ QueueReceiveW\_TQ$	
	$\lor\ QueueReceiveWS\_TQ$	
	$\lor\ QueueReceiveF\_TQ$	
	$QueueReceiveN\_TQ$	$que? \in queue$
		$q\_size(que?) > 0$
		$que? \notin \operatorname{ran} wait\_snd$
	$QueueReceiveW\_TQ$	$que? \in queue$
		$q\_size(que?) > 0$
		$\forall\, ws : wait\_snd^{\sim}(\!\mid \{que?\} \mid\!) \bullet priority(running\_task) \geq priority(ws)$
	$QueueReceiveWS\_TQ$	$que? \in queue$
		$q\_size(que?) > 0$
		$\exists\, topReady! : wait\_snd^{\sim}(\!\mid \{que?\} \mid\!)$
		$\quad \mid (\forall\, ws : wait\_snd^{\sim}(\!\mid \{que?\} \mid\!) \bullet priority(topReady!) \geq priority(ws))$
		$\quad \bullet priority(topReady!) > priority(running\_task)$
	$QueueReceiveE\_TQ$	$que? \in queue$
		$q\_size(que?) = 0$
		$running\_task \neq idle$
vTaskDelayUntil	$DelayUntil\_TQT$	$n? > clock$

Table C.6: API mappings & preconditions for operations(continue)

vSemaphoreCreateBinary	$CreateBinarySemaphore\_TQTM$	$sem? \notin queue$
vSemaphoreDelete	$DeleteBinarySemaphore\_TQTM$	$sem? \in semaphore$
xSemaphoreCreateMutex	$CreateMutex\_TQTM$	$mut? \notin queue$
vSemaphoreDelete	$DeleteMutex\_TQTM$	$mut? \in mutex \setminus \mathrm{dom}\, mutex\_holder$
xSemaphoreTake	$MutexTake\_TQTM$	$mut? \in mutex$
	$\widehat{=} MutexTakeNnonInh\_TQTM$	$q\_size(mut?) = 0 \Rightarrow$
	$\lor MutexTakeNInh\_TQTM$	$(running\_task \neq idle$
	$\lor MutexTakeEnonInh\_TQTM$	$\land\ n? > clock$
	$\lor MutexTakeEInheritReady\_TQTM$	$\land\ mutex\_holder(mut?) \neq idle)$
	$\lor MutexTakeEInheritHolder\_TQTM$	
	$\lor MutexTakeRecursive\_TQTM$	
	$MutexTakeNnonInh\_TQTM$	$mut? \in mutex$
		$q\_size(mut?) > 0$
		$running\_task \notin \mathrm{dom}\, base\_priority$
	$MutexTakeNInh\_TQTM$	$mut? \in mutex$
		$q\_size(mut?) > 0$
		$running\_task \in \mathrm{dom}\, base\_priority$
	$MutexTakeEnonInh\_TQTM$	$mut? \in \mathrm{dom}\, mutex\_holder$
		$priority(running\_task) \leq priority(mutex\_holder(mut?))$
		$n? > clock$
		$running\_task \neq idle$
		$running\_task \neq mutex\_holder(mut?)$

Table C.7: API mappings & preconditions for operations(continue)

	$MutexTakeEInheritReady\_TQTM$	$mut? \in \mathrm{dom}\, mutex\_holder$ $priority(running\_task) > priority(mutex\_holder(mut?))$ $n? > clock$ $mutex\_holder(mut?) \neq idle$ $state(mutex\_holder(mut?)) \neq ready$
	$MutexTakeEInheritHolder\_TQTM$	$mut? \in \mathrm{dom}\, mutex\_holder$ $priority(running\_task) > priority(mutex\_holder(mut?))$ $n? > clock$ $mutex\_holder(mut?) \neq idle$ $state(mutex\_holder(mut?)) = ready$
	$MutexTakeRecursive\_TQTM$	$mut? \in \mathrm{dom}\, mutex\_holder$ $running\_task = mutex\_holder(mut?)$
xSemaphoreGive	$MutexGive\_TQTM$ $\widehat{=} MutexGiveNRecursive\_TQTM$ $\vee\ MutexGiveNnonInh\_TQTM$ $\vee\ MutexGiveNInhN\_TQTM$ $\vee\ MutexGiveNInhS\_TQTM$ $\vee\ MutexGiveWnonInhN\_TQTM$ $\vee\ MutexGiveWnonInhS\_TQTM$ $\vee\ MutexGiveWInhN\_TQTM$ $\vee\ MutexGiveWInhSR\_TQTM$ $\vee\ MutexGiveWInhSW\_TQTM$	$mut? \in \mathrm{dom}\, mutex\_holder$ $running\_task = mutex\_holder(mut?)$

Table C.8: API mappings & preconditions for operations(continue)

$MutexGiveNRecursive\_TQTM$	$mut? \in \mathrm{dom}\, mutex\_holder$ $running\_task = mutex\_holder(mut?)$ $mutex\_recursive(mut?) > 1$
$MutexGiveNnonInh\_TQTM$	$mut? \in \mathrm{dom}\, mutex\_holder$ $mut? \notin \mathrm{ran}\, wait\_rcv$ $running\_task = mutex\_holder(mut?)$ $mutex\_recursive(mut?) = 1$ $base\_priority(running\_task) = priority(running\_task)$
$MutexGiveNInhN\_TQTM$	$mut? \in \mathrm{dom}\, mutex\_holder$ $running\_task = mutex\_holder(mut?)$ $mutex\_recursive(mut?) = 1$ $mut? \notin \mathrm{ran}\, wait\_rcv$ $base\_priority(running\_task) \neq priority(running\_task)$ $\forall\, rt : state^{\sim}(\!\mid \{ready\} \mid\!) \bullet base\_priority(running\_task) \geq priority(rt)$
$MutexGiveNInhS\_TQTM$	$mut? \in \mathrm{dom}\, mutex\_holder$ $running\_task = mutex\_holder(mut?)$ $mutex\_recursive(mut?) = 1$ $mut? \notin \mathrm{ran}\, wait\_rcv$ $base\_priority(running\_task) \neq priority(running\_task)$ $\exists\, topReady! : state^{\sim}(\!\mid \{ready\} \mid\!) \mid \forall\, rt : state^{\sim}(\!\mid \{ready\} \mid\!)$ $\bullet\ priority(topReady!) \geq priority(rt)$ $\bullet\ base\_priority(running\_task) < priority(topReady!)$

Table C.9: API mappings & preconditions for operations(continue)

$MutexGiveWnonInhN\_TQTM$	$mut? \in \mathrm{dom}\ mutex\_holder$ $running\_task = mutex\_holder(mut?)$ $mutex\_recursive(mut?) = 1$ $base\_priority(running\_task) = priority(running\_task)$ $\forall\, wr : wait\_rcv^{\sim}(\!\vert\ \{mut?\}\ \vert\!) \bullet priority(running\_task) \geq priority(wr)$
$MutexGiveWnonInhS\_TQTM$	$mut? \in \mathrm{dom}\ mutex\_holder$ $running\_task = mutex\_holder(mut?)$ $mutex\_recursive(mut?) = 1$ $base\_priority(running\_task) = priority(running\_task)$ $\exists\, topReady! : wait\_rcv^{\sim}(\!\vert\ \{mut?\}\ \vert\!)$ $\quad \vert\ (\forall\, wr : wait\_rcv^{\sim}(\!\vert\ \{mut?\}\ \vert\!) \bullet priority(topReady!) \geq priority(wr))$ $\quad \bullet\ priority(topReady!) > priority(running\_task)$
$MutexGiveWInhN\_TQTM$	$mut? \in \mathrm{dom}\ mutex\_holder$ $running\_task = mutex\_holder(mut?)$ $mutex\_recursive(mut?) = 1$ $\forall\, wr : wait\_rcv^{\sim}(\!\vert\ \{mut?\}\ \vert\!) \bullet base\_priority(running\_task) \geq priority(wr)$ $base\_priority(running\_task) \neq priority(running\_task)$ $\forall\, rt : state^{\sim}(\!\vert\ \{ready\}\ \vert\!) \bullet base\_priority(running\_task) \geq priority(rt)$

Table C.10: API mappings & preconditions for operations(continue)

$MutexGiveWInhSR\_TQTM$	$mut? \in \mathrm{dom}\, mutex\_holder$ $running\_task = mutex\_holder(mut?)$ $mutex\_recursive(mut?) = 1$ $base\_priority(running\_task) \neq priority(running\_task)$ $\exists\, topReady! : state^{\sim}(\!\mid \{ready\} \mid\!)$ $\bullet\ (\forall\, rt : state^{\sim}(\!\mid \{ready\} \mid\!) \bullet priority(topReady!) \geq priority(rt))$ $\wedge\ (\forall\, wr : wait\_rcv^{\sim}(\!\mid \{mut?\} \mid\!)$ $\bullet\ priority(topReady!) > priority(wr))$ $\wedge\ priority(topReady!) > base\_priority(running\_task)$
$MutexGiveWInhSW\_TQTM$	$mut? \in \mathrm{dom}\, mutex\_holder$ $running\_task = mutex\_holder(mut?)$ $mutex\_recursive(mut?) = 1$ $base\_priority(running\_task) \neq priority(running\_task)$ $\exists\, topWaiting! : wait\_rcv^{\sim}(\!\mid \{mut?\} \mid\!)$ $\bullet\ (\forall\, wr : wait\_rcv^{\sim}(\!\mid \{mut?\} \mid\!) \bullet priority(topWaiting!) \geq priority(wr))$ $\wedge\ (\forall\, rt : state^{\sim}(\!\mid \{ready\} \mid\!)$ $\bullet\ priority(topWaiting!) \geq priority(rt))$ $\wedge\ priority(topWaiting!) > base\_priority(running\_task)$

Appendix D

SPECIFICATION FOR TASK MODEL

$[CONTEXT, TASK]$

$$
\begin{array}{|l}
bare\_context : CONTEXT \\
idle : TASK
\end{array}
$$

$$STATE ::= nonexistent \mid ready \mid blocked \mid suspended \mid running$$

$$
\begin{aligned}
transition == & (\{blocked\} \times \{nonexistent, ready, running, suspended\}) \\
& \cup \ (\{nonexistent\} \times \{ready, running\}) \\
& \cup \ (\{ready\} \times \{nonexistent, running, suspended\}) \\
& \cup \ (\{running\} \times \{blocked, nonexistent, ready, suspended\}) \\
& \cup \ (\{suspended\} \times \{nonexistent, ready, running\})
\end{aligned}
$$

theorem grule gTransitionType
$transition \in \mathbb{P}(STATE \times STATE)$

theorem rule lInTransition
$\forall\, l, r : STATE \mid (l, r) \in \{nonexistent \mapsto ready, running \mapsto ready,$
$blocked \mapsto ready, suspended \mapsto ready, ready \mapsto running,$
$blocked \mapsto running, suspended \mapsto running,$
$nonexistent \mapsto running, running \mapsto suspended,$
$ready \mapsto suspended, blocked \mapsto suspended,$
$running \mapsto blocked, running \mapsto nonexistent,$
$ready \mapsto nonexistent, blocked \mapsto nonexistent,$
$suspended \mapsto nonexistent\}$
$\bullet\ (l, r) \in transition$

TaskData

$tasks : \mathbb{F}\ TASK$
$running\_task : TASK$

$running\_task \in tasks$
$idle \in tasks$

Init_TaskData

$TaskData'$

$tasks' = \{idle\}$
$running\_task' = idle$

theorem TaskDataInit
$\exists\ TaskData' \bullet Init\_TaskData$

StateData

$state : TASK \rightarrow STATE$

$state(idle) \in \{ready, running\}$

Init_StateData

$StateData'$

$state' = (\lambda\, x : TASK \bullet nonexistent) \oplus \{(idle \mapsto running)\}$

theorem StateDataInit
$\exists\ StateData' \bullet Init\_StateData$

ContextData

$phys\_context : CONTEXT$
$log\_context : TASK \rightarrow CONTEXT$

Init_ContextData

$ContextData'$

$phys\_context' = bare\_context$
$log\_context' = (\lambda\, x : TASK \bullet bare\_context)$

theorem ContextDataInit

$\exists\, ContextData' \bullet Init\_ContextData$

PrioData

$priority : TASK \to \mathbb{N}$

$priority(idle) = 0$

Init_PrioData

$PrioData'$

$priority' = (\lambda\, x : TASK \bullet 0)$

theorem PrioDataInit

$\exists\, PrioData' \bullet Init\_PrioData$

Task

$TaskData$
$StateData$
$ContextData$
$PrioData$

$tasks = TASK \setminus (state^{\sim}(\!|\, \{nonexistent\} \,|\!))$
$state^{\sim}(\!|\, \{running\} \,|\!) = \{running\_task\}$
$\forall\, pt : state^{\sim}(\!|\, \{ready\} \,|\!) \bullet priority(running\_task) \geq priority(pt)$

$\Delta\, Task$

$Task$
$Task'$

$\forall\, st : TASK \mid state'(st) \neq state(st)$
$\quad \bullet\ state(st) \mapsto state'(st) \in transition$

$f : \mathbb{P}\, TASK \to TASK$

$\langle\!\langle$ findDelegate $\rangle\!\rangle$
$\forall\, Task;\ a : \mathbb{P}\, TASK;\ g : TASK \nrightarrow \mathbb{Z} \bullet$
$f(a) \in a \land a \subseteq \operatorname{dom} g \land$
$(\forall\, t : a \bullet g(f(a)) \geq g(t))$

theorem TaskProperty1

$\forall\, Task \bullet state(running\_task) = running$

theorem TaskProperty2

$$\forall\, Task \bullet \forall\, t : TASK \mid t \in state^{\sim}(\!|\, \{blocked\} \,|\!) \bullet t \in tasks$$

theorem TaskProperty3

$$\forall\, Task \bullet \forall\, t : state^{\sim}(\!|\, \{ready\} \,|\!) \bullet t \in tasks \setminus \{running\_task\}$$

theorem TaskProperty6

$$\forall\, Task;\ t : TASK \mid 0 < priority(t) \bullet idle \neq t$$

Init_Task

$$Task'$$

$$Init\_TaskData$$
$$Init\_StateData$$
$$Init\_ContextData$$
$$Init\_PrioData$$

theorem TaskInit

$$\exists\, Task' \bullet Init\_Task$$

Reschedule

$$\Delta Task$$
$$target? : TASK$$
$$tasks? : \mathbb{F}\ TASK$$
$$st? : STATE$$
$$pri? : TASK \to \mathbb{N}$$

$$tasks' = tasks?$$
$$running\_task' = target?$$
$$state' = state \oplus \{(target? \mapsto running), (running\_task \mapsto st?)\}$$
$$phys\_context' = log\_context(target?)$$
$$log\_context' = log\_context \oplus \{(running\_task \mapsto phys\_context)\}$$
$$priority' = pri?$$

$$disableReschedule \mathrel{\widehat=} [Task \mid false] \land Reschedule$$

CreateTaskN_T

$$\Delta Task$$
$$target? : TASK$$
$$newpri? : \mathbb{N}$$

$$state(target?) = nonexistent$$

$$
\begin{array}{l}
newpri? \leq priority(running\_task) \\
tasks' = tasks \cup \{target?\} \\
running\_task' = running\_task \\
state' = state \oplus \{(target? \mapsto ready)\} \\
\Xi ContextData \\
priority' = priority \oplus \{(target? \mapsto newpri?)\}
\end{array}
$$

CreateTaskN_TFSBSig

$$
\begin{array}{l}
Task \\
target? : TASK \\
newpri? : \mathbb{N} \\
\hline
state(target?) = nonexistent \\
newpri? \leq priority(running\_task)
\end{array}
$$

theorem rule runningUpdate

$$
\begin{array}{l}
\forall f : TASK \to STATE;\ g : TASK \pfun STATE \mid running \notin \operatorname{ran} g \\
\qquad \land (f^{\sim}(\!| \{running\} |\!)) \cap \operatorname{dom} g = \emptyset \\
\qquad \bullet (f \oplus g)^{\sim}(\!| \{running\} |\!) = f^{\sim}(\!| \{running\} |\!)
\end{array}
$$

theorem rule setminUpdate

$$
\begin{array}{l}
\forall f : TASK \to STATE;\ g : TASK \pfun STATE \\
\qquad \bullet\ TASK \setminus ((f \oplus g)^{\sim}(\!| \{nonexistent\} |\!)) \\
\qquad\qquad = TASK \setminus (f^{\sim}(\!| \{nonexistent\} |\!)) \setminus (g^{\sim}(\!| \{nonexistent\} |\!)) \\
\qquad\qquad \cup (\operatorname{dom} g \setminus (g^{\sim}(\!| \{nonexistent\} |\!)))
\end{array}
$$

theorem CreateTaskN_T_vc_ref

$$
\forall\ CreateTaskN\_TFSBSig \mid true \bullet \operatorname{pre} CreateTaskN\_T
$$

CreateTaskS_T

$$
\begin{array}{l}
\Delta Task \\
target? : TASK \\
newpri? : \mathbb{N} \\
\hline
state(target?) = nonexistent \\
newpri? > priority(running\_task) \\
\exists\, st? : STATE;\ tasks? : \mathbb{F}\ TASK;\ pri? : TASK \to \mathbb{N} \\
\qquad \mid st? = ready \land tasks? = tasks \cup \{target?\} \\
\qquad \land pri? = priority \oplus \{(target? \mapsto newpri?)\} \\
\qquad \bullet\ Reschedule
\end{array}
$$

CreateTaskS_TFSBSig

$Task$
$target? : TASK$
$newpri? : \mathbb{N}$

$state(target?) = nonexistent$
$newpri? > priority(running\_task)$

theorem CreateTaskS_T_vc_ref
$\forall\, CreateTaskS\_TFSBSig \mid true \bullet \text{pre}\ CreateTaskS\_T$

$CreateTask\_T \mathrel{\widehat=} CreateTaskN\_T \lor CreateTaskS\_T$

DeleteTaskN_T

$\Delta Task$
$target? : TASK$
$topReady! : TASK$

$target? \in tasks \setminus \{idle\}$
$state(target?) \in \{ready, blocked, suspended\}$
$tasks' = tasks \setminus \{target?\}$
$running\_task' = running\_task$
$state' = state \oplus \{(target? \mapsto nonexistent)\}$
$phys\_context' = phys\_context$
$log\_context' = log\_context \oplus \{(target? \mapsto bare\_context)\}$
$\Xi PrioData$
$topReady! = running\_task$

DeleteTaskN_TFSBSig

$Task$
$target? : TASK$

$target? \in tasks \setminus \{idle\}$
$state(target?) \in \{ready, blocked, suspended\}$

theorem finsetIsFinset
$\forall\, X : \mathbb{F}\ TASK;\ x : TASK \bullet X \setminus \{x\} \in \mathbb{F}\ TASK$

theorem DeleteTaskN_T_vc_ref
$\forall\, DeleteTaskN\_TFSBSig \mid true \bullet \text{pre}\ DeleteTaskN\_T$

DeleteTaskS_T

$\Delta Task$
$target? : TASK$
$topReady! : TASK$

$target? \in tasks \setminus \{idle\}$
$state(target?) \in \{running\}$
$state(topReady!) = ready$
$\forall t : state^{\sim}(\!| \{ready\} |\!) \bullet priority(topReady!) \geq priority(t)$
$tasks' = tasks \setminus \{target?\}$
$running\_task' = topReady!$
$state' = state \oplus \{(topReady! \mapsto running), (target? \mapsto nonexistent)\}$
$phys\_context' = log\_context(topReady!)$
$log\_context' = log\_context \oplus \{(target? \mapsto bare\_context)\}$
$\Xi PrioData$

DeleteTaskS_TFSBSig

$Task$
$target? : TASK$

$target? \in tasks \setminus \{idle\}$
$state(target?) \in \{running\}$

theorem lDeleteTaskS_T_Lemma

$\forall Task;\ topReady!, target? : TASK$
$\quad | \ target? \in tasks \setminus \{idle\}$
$\quad \land \ state(target?) \in \{running\}$
$\quad \land \ state(topReady!) = ready$
$\quad \land \ (\forall rtsk : state^{\sim}(\!| \{ready\} |\!) \bullet priority(topReady!) \geq priority(rtsk))$
$\quad \bullet \neg \ (Task[log\_context := log\_context \oplus \{(target?, bare\_context)\},$
$\qquad phys\_context := log\_context(topReady!),$
$\qquad running\_task := topReady!,$
$\qquad state := state \oplus$
$\qquad\quad (\{(target?, nonexistent)\} \cup \{(topReady!, running)\}),$
$\qquad tasks := tasks \setminus \{target?\}]$
$\qquad \land \ (st \in TASK$
$\qquad\quad \land \neg \ (state \oplus (\{(target?, nonexistent)\} \cup$
$\qquad\qquad \{(topReady!, running)\}))(st) = state(st)$
$\qquad\quad \Rightarrow (state(st), (state \oplus (\{(target?, nonexistent)\} \cup$
$\qquad\qquad \{(topReady!, running)\}))(st)) \in transition)$
$\qquad \Rightarrow t \in TASK$
$\qquad\quad \land \ state(t) = ready$
$\qquad\quad \land \neg \ priority(topReady!) \geq priority(t))$

theorem DeleteTaskS_T_vc_ref

$\forall\, DeleteTaskS\_TFSBSig \mid true \bullet \text{pre}\ DeleteTaskS\_T$

$DeleteTask\_T \mathrel{\widehat=} DeleteTaskN\_T \lor DeleteTaskS\_T$

ExecuteRunningTask_T

$\Delta Task$

$target! : TASK$

$\Xi TaskData$

$\Xi StateData$

$phys\_context' \neq phys\_context$

$log\_context' = log\_context$

$\Xi PrioData$

$target! = running\_task$

ExecuteRunningTask_TFSBSig

$Task$

$\exists\, phys\_context' : CONTEXT \bullet phys\_context' \neq phys\_context$

theorem ExecuteRunningTask_T_vc_ref

$\forall\, ExecuteRunningTask\_TFSBSig \mid true \bullet \text{pre}\ ExecuteRunningTask\_T$

SuspendTaskN_T

$\Delta Task$

$target? : TASK$

$topReady! : TASK$

$target? \in tasks \setminus \{idle\}$

$state(target?) \in \{ready, blocked\}$

$\Xi TaskData$

$state' = state \oplus \{(target? \mapsto suspended)\}$

$\Xi ContextData$

$\Xi PrioData$

$topReady! = running\_task$

SuspendTaskN_TFSBSig

$Task$

$target? : TASK$

$target? \in tasks \setminus \{idle\}$

$state(target?) \in \{ready, blocked\}$

theorem SuspendTaskN_T_vc_ref

$\forall\, SuspendTaskN\_TFSBSig \mid true \bullet \text{pre}\ SuspendTaskN\_T$

SuspendTaskS_T

$\Delta Task$
$target? : TASK$
$topReady! : TASK$

$target? \in tasks \setminus \{idle\}$
$state(target?) \in \{running\}$
$state(topReady!) = ready$
$\forall\, t : state^{\sim}(\!|\ \{ready\}\ |\!) \bullet priority(topReady!) \geq priority(t)$
$\exists\, st? : STATE \mid st? = suspended$
$\quad \bullet\ Reschedule[tasks/tasks?, priority/pri?, topReady!/target?]$

theorem TaskProperty4

$\forall\, Task \mid SuspendTaskS\_T$
$\quad \bullet\ state'(running\_task) = suspended$
$\quad \wedge\ (\forall\, t : state^{\sim}(\!|\ \{ready\}\ |\!) \bullet priority(running\_task') \geq priority(t))$

theorem TaskProperty5

$\forall\, Task \bullet \forall\, t : TASK \mid t \notin tasks \bullet state(t) = nonexistent$

SuspendTaskS_TFSBSig

$Task$
$target? : TASK$

$target? \in tasks \setminus \{idle\}$
$state(target?) \in \{running\}$

theorem lSuspendTaskS_T_Lemma

$\forall\, Task;\ target?, topReady! : TASK$
$\quad \mid target? \in tasks \setminus \{idle\}$
$\quad \wedge\ state(target?) \in \{running\}$
$\quad \wedge\ state(topReady!) = ready$
$\quad \wedge\ (\forall\, rtsk : state^{\sim}(\!|\ \{ready\}\ |\!) \bullet priority(topReady!) \geq priority(rtsk))$
$\quad \bullet\ \neg\ (Task[log\_context := log\_context \oplus$
$\qquad\qquad \{(running\_task, phys\_context)\},$
$\qquad\quad phys\_context := log\_context(topReady!),$
$\qquad\quad running\_task := topReady!,$

$$
\begin{array}{l}
\quad state := state \oplus \\
\qquad (\{(running\_task, suspended)\} \cup \{(topReady!, running)\})] \\
\wedge\ (st \in TASK \\
\quad \wedge \neg\ (state \oplus (\{(running\_task, suspended)\} \\
\qquad \cup \{(topReady!, running)\}))st = state(st) \\
\qquad \Rightarrow (state(st), (state \oplus (\{(running\_task, suspended)\} \\
\qquad \cup \{(topReady!, running)\}))st) \in transition) \\
\Rightarrow t \in TASK \\
\quad \wedge\ state(t) = ready \\
\quad \wedge \neg\ priority(topReady!) \geq priority(t))
\end{array}
$$

theorem SuspendTaskS_T_vc_ref

$\forall\ SuspendTaskS\_TFSBSig \mid true \bullet \text{pre}\ SuspendTaskS\_T$

SuspendTaskO_T

$\Xi Task$

$target? : TASK$

$topReady! : TASK$

$state(target?) \in \{suspended\}$

$topReady! = running\_task$

SuspendTaskO_TFSBSig

$Task$

$target? : TASK$

$topReady! : TASK$

$state(target?) \in \{suspended\}$

theorem SuspendTaskO_T_vc_ref

$\forall\ SuspendTaskO\_TFSBSig \mid true \bullet \text{pre}\ SuspendTaskO\_T$

$$
\begin{array}{l}
SuspendTask\_T \mathrel{\widehat=} SuspendTaskN\_T \\
\qquad\qquad \vee\ SuspendTaskS\_T \\
\qquad\qquad \vee\ SuspendTaskO\_T
\end{array}
$$

ResumeTaskN_T

$\Delta Task$

$target? : TASK$

$state(target?) = suspended$

$priority(target?) \leq priority(running\_task)$

$\Xi TaskData$

$state' = state \oplus \{(target? \mapsto ready)\}$
$\Xi ContextData$
$\Xi PrioData$

ResumeTaskN_TFSBSig
$Task$
$target? : TASK$

$state(target?) = suspended$
$priority(target?) \leq priority(running\_task)$

theorem ResumeTaskN_T_vc_ref
$\forall ResumeTaskN\_TFSBSig \mid true \bullet \text{pre } ResumeTaskN\_T$

ResumeTaskS_T
$\Delta Task$
$target? : TASK$

$state(target?) = suspended$
$priority(target?) > priority(running\_task)$
$\exists st? : STATE \mid st? = ready$
$\bullet Reschedule[tasks/tasks?, priority/pri?]$

ResumeTaskS_TFSBSig
$Task$
$target? : TASK$

$state(target?) = suspended$
$priority(target?) > priority(running\_task)$

theorem ResumeTaskS_T_vc_ref
$\forall ResumeTaskS\_TFSBSig \mid true \bullet \text{pre } ResumeTaskS\_T$

$ResumeTask\_T \mathrel{\widehat=} ResumeTaskN\_T \lor ResumeTaskS\_T$

ChangeTaskPriorityN_T
$\Delta Task$
$newpri? : \mathbb{N}$
$target? : TASK$
$topReady! : TASK$

$state(target?) = ready \Rightarrow newpri? \leq priority(running\_task)$
$state(target?) = running \Rightarrow$
$\quad (\forall t : state^{\sim}(\!| \{ready\} |\!) \bullet newpri? \geq priority(t))$
$state(target?) \neq nonexistent$
$target? = idle \Rightarrow newpri? = 0$
$\Xi TaskData$
$\Xi StateData$
$\Xi ContextData$
$priority' = priority \oplus \{(target? \mapsto newpri?)\}$
$topReady! = running\_task$

ChangeTaskPriorityN_TFSBSig

$Task$
$newpri? : \mathbb{N}$
$target? : TASK$

$state(target?) = ready \Rightarrow newpri? \leq priority(running\_task)$
$state(target?) = running \Rightarrow$
$\quad (\forall t : state^{\sim}(\!| \{ready\} |\!) \bullet newpri? \geq priority(t))$
$state(target?) \neq nonexistent$
$target? = idle \Rightarrow newpri? = 0$

theorem ChangeTaskPriorityN_T_vc_ref
$\forall ChangeTaskPriorityN\_TFSBSig \mid true \bullet \text{pre } ChangeTaskPriorityN\_T$

ChangeTaskPriorityS_T

$\Delta Task$
$target? : TASK$
$newpri? : \mathbb{N}$
$topReady! : TASK$

$state(target?) = ready$
$newpri? > priority(running\_task)$
$target? = idle \Rightarrow newpri? = 0$
$\exists st? : STATE;\ pri? : TASK \rightarrow \mathbb{N}$
$\quad \mid st? = ready$
$\quad\quad \wedge\ pri? = priority \oplus \{(target? \mapsto newpri?)\}$
$\quad \bullet Reschedule[tasks/tasks?]$
$topReady! = target?$

ChangeTaskPriorityS_TFSBSig

$Task$
$newpri? : \mathbb{N}$

$target? : TASK$

$state(target?) = ready$
$newpri? > priority(running\_task)$
$target? = idle \Rightarrow newpri? = 0$

theorem ChangeTaskPriorityS_T_vc_ref
$\forall\, ChangeTaskPriorityS\_TFSBSig \mid true \bullet \text{pre}\ ChangeTaskPriorityS\_T$

ChangeTaskPriorityD_T

$\Delta Task$
$target? : TASK$
$topReady! : TASK$
$newpri? : \mathbb{N}$

$state(target?) = running$
$target? = idle \Rightarrow newpri? = 0$
$state(topReady!) = ready$
$\forall\, t : state^{\sim}(\!|\, \{ready\} \,|\!) \bullet priority(topReady!) \geq priority(t)$
$newpri? < priority(topReady!)$
$\exists\, st? : STATE;\ pri? : TASK \to \mathbb{N}$
$\quad \mid st? = ready$
$\quad\quad \land\ pri? = priority \oplus \{(target? \mapsto newpri?)\}$
$\quad \bullet\ Reschedule[tasks/tasks?, topReady!/target?]$

ChangeTaskPriorityD_TFSBSig

$Task$
$newpri? : \mathbb{N}$
$target? : TASK$

$state(target?) = running$
$target? = idle \Rightarrow newpri? = 0$
$\exists\, readyTask : state^{\sim}(\!|\, \{ready\} \,|\!) \bullet newpri? < priority(readyTask)$

theorem lChangeTaskPriorityD_T_Lemma
$\forall\, Task;\ target?, topReady! : TASK;\ newpri? : \mathbb{N}$
$\quad \mid state(target?) = running$
$\quad \land\ (target? = idle \Rightarrow newpri? = 0)$
$\quad \land\ state(topReady!) = ready$
$\quad \land\ (\forall\, rtsk : state^{\sim}(\!|\, \{ready\} \,|\!) \bullet priority(topReady!) \geq priority(rtsk))$
$\quad \land\ newpri? < priority(topReady!)$
$\quad \bullet \neg\ (Task[log\_context := log\_context \oplus$
$\quad\quad\quad\quad \{(running\_task, phys\_context)\},$

$$
\begin{array}{l}
\quad phys\_context := log\_context(topReady!), \\
\quad priority := priority \oplus \{(target?, newpri?)\}, \\
\quad running\_task := topReady!, \\
\quad state := state \oplus \\
\qquad (\{(running\_task, ready)\} \cup \{(topReady!, running)\})] \\
\wedge\ (st \in TASK \\
\quad \wedge \neg\ (state \oplus (\{(running\_task, ready)\} \cup \\
\qquad \{(topReady!, running)\}))st = state(st) \\
\qquad \Rightarrow (state(st), (state \oplus (\{(running\_task, ready)\} \cup \\
\qquad \{(topReady!, running)\}))st) \in transition) \\
\Rightarrow t \in TASK \\
\quad \wedge\ state(t) = ready \\
\quad \wedge \neg\ priority(topReady!) \geq priority(t))
\end{array}
$$

theorem ChangeTaskPriorityD_T_vc_ref

$\forall\ ChangeTaskPriorityD\_TFSBSig \mid true \bullet \text{pre}\ ChangeTaskPriorityD\_T$

$$
\begin{array}{l}
ChangeTaskPriority\_T \mathrel{\widehat=} ChangeTaskPriorityN\_T \\
\qquad\qquad\qquad\qquad \vee\ ChangeTaskPriorityS\_T \\
\qquad\qquad\qquad\qquad \vee\ ChangeTaskPriorityD\_T
\end{array}
$$

Appendix E

SPECIFICATION FOR QUEUE MODEL

$[QUEUE]$

QueueData

$$queue : \mathbb{P}\, QUEUE$$
$$q\_max : QUEUE \pfun \mathbb{N}_1$$
$$q\_size : QUEUE \pfun \mathbb{N}$$

$$\mathrm{dom}\, q\_max = \mathrm{dom}\, q\_size = queue$$
$$\forall\, q : QUEUE \mid q \in queue \bullet q\_size(q) \leq q\_max(q)$$

Init_QueueData

$$QueueData'$$

$$queue' = \emptyset$$
$$q\_max' = \emptyset$$
$$q\_size' = \emptyset$$

theorem QueueDataInit

$$\exists\, QueueData' \bullet Init\_QueueData$$

WaitingData

$$wait\_snd : TASK \pfun QUEUE$$
$$wait\_rcv : TASK \pfun QUEUE$$

$$\mathrm{dom}\, wait\_snd \cap \mathrm{dom}\, wait\_rcv = \emptyset$$

Init_WaitingData

$$WaitingData'$$

$$wait\_snd' = \emptyset$$
$$wait\_rcv' = \emptyset$$

theorem WaitingDataInit

$$\exists\, WaitingData' \bullet Init\_WaitingData$$

QReleasingData

$$release\_snd : TASK \pfun QUEUE$$
$$release\_rcv : TASK \pfun QUEUE$$

$$\mathrm{dom}\, release\_snd \cap \mathrm{dom}\, release\_rcv = \emptyset$$

Init_QReleasingData

$$QReleasingData'$$

$$release\_snd' = \emptyset$$
$$release\_rcv' = \emptyset$$

theorem QReleasingDataInit

$$\exists\, QReleasingData' \bullet Init\_QReleasingData$$

Queue

$$QueueData$$
$$WaitingData$$
$$QReleasingData$$

$$\mathrm{ran}\, wait\_snd \subseteq queue$$
$$\mathrm{ran}\, wait\_rcv \subseteq queue$$
$$\mathrm{ran}\, release\_snd \subseteq queue$$
$$\mathrm{ran}\, release\_rcv \subseteq queue$$
$$(\mathrm{dom}\, wait\_snd \cup \mathrm{dom}\, wait\_rcv)$$
$$\quad \cap (\mathrm{dom}\, release\_snd \cup \mathrm{dom}\, release\_rcv) = \emptyset$$

theorem QueueProperty1

$$\forall\, Queue \bullet \forall\, q : queue \bullet q\_max(q) > 0$$

theorem ImageProperty1

$$\forall\, f : TASK \pfun QUEUE \bullet \forall\, y : QUEUE \bullet f^{\sim}(\!|\, \{y\} \,|\!) \neq \emptyset \Leftrightarrow y \in \mathrm{ran}\, f$$

Init_Queue

$Queue'$

$Init\_QueueData$
$Init\_WaitingData$
$Init\_QReleasingData$

theorem QueueInit
$\exists\, Queue' \bullet Init\_Queue$

TaskQueue

$Task$
$Queue$

$\mathrm{dom}\, wait\_snd \subseteq state^{\sim}(\!|\, \{blocked\} \,|\!)$
$\mathrm{dom}\, wait\_rcv \subseteq state^{\sim}(\!|\, \{blocked\} \,|\!)$

Init_TaskQueue

$TaskQueue'$

$Init\_Task$
$Init\_Queue$

theorem TaskQueueInit
$\exists\, TaskQueue' \bullet Init\_TaskQueue$

$\Delta TaskQueue \mathrel{\widehat=} TaskQueue \land TaskQueue' \land \Delta Task$

ExtendTaskXi

$\Delta TaskQueue$

$running\_task \notin \mathrm{dom}\, release\_snd \cup \mathrm{dom}\, release\_rcv$
$\Xi Queue$

ExtTaskFSBSig

$TaskQueue$

$running\_task \notin \mathrm{dom}\, release\_snd \cup \mathrm{dom}\, release\_rcv$

$CreateTaskN\_TQ \mathrel{\widehat=} ExtendTaskXi \land CreateTaskN\_T$

$CreateTaskN\_TQFSBSig \mathrel{\widehat=} ExtTaskFSBSig \land CreateTaskN\_TFSBSig$

theorem CreateTaskN_TQ_vc_ref
$\forall\, CreateTaskN\_TQFSBSig \mid true \bullet \text{pre}\ CreateTaskN\_TQ$

$CreateTaskS\_TQ \mathrel{\widehat=} ExtendTaskXi \land CreateTaskS\_T$

$CreateTaskS\_TQFSBSig \mathrel{\widehat=} ExtTaskFSBSig \land CreateTaskS\_TFSBSig$

theorem CreateTaskS_TQ_vc_ref
$\forall\, CreateTaskS\_TQFSBSig \mid true \bullet \text{pre}\ CreateTaskS\_TQ$

$CreateTask\_TQ \mathrel{\widehat=} CreateTaskN\_TQ \lor CreateTaskS\_TQ$

DeleteTaskN_TQ

$DeleteTaskN\_T$
$\Delta TaskQueue$

$running\_task \notin \text{dom}\ release\_snd \cup \text{dom}\ release\_rcv$
$\Xi QueueData$
$wait\_snd' = \{target?\} \lhd wait\_snd$
$wait\_rcv' = \{target?\} \lhd wait\_rcv$
$release\_snd' = \{target?\} \lhd release\_snd$
$release\_rcv' = \{target?\} \lhd release\_rcv$

$DeleteTaskN\_TQFSBSig \mathrel{\widehat=} ExtTaskFSBSig \land DeleteTaskN\_TFSBSig$

theorem DeleteTaskN_TQ_vc_ref
$\forall\, DeleteTaskN\_TQFSBSig \mid true \bullet \text{pre}\ DeleteTaskN\_TQ$

$DeleteTaskS\_TQ \mathrel{\widehat=} ExtendTaskXi \land DeleteTaskS\_T$

$DeleteTaskS\_TQFSBSig \mathrel{\widehat=} ExtTaskFSBSig \land DeleteTaskS\_TFSBSig$

theorem lDeleteTaskS_TQ_Lemma
$\forall\, TaskQueue;\ topReady!, target? : TASK$
$\quad \mid running\_task \notin \text{dom}\ release\_snd \cup \text{dom}\ release\_rcv$
$\quad \land\ target? \in tasks \setminus \{idle\}$
$\quad \land\ state(target?) \in \{running\}$
$\quad \land\ state(topReady!) = ready$

$$
\begin{array}{l}
\quad \wedge\ (\forall\, rtsk : state^{\sim}(\!|\ \{ready\}\ |\!) \bullet priority(topReady!) \geq priority(rtsk)) \\
\quad \bullet \neg\ (TaskQueue[log\_context := log\_context \oplus \\
\qquad\qquad \{(target?, bare\_context)\}, \\
\qquad\quad phys\_context := log\_context(topReady!), \\
\qquad\quad running\_task := topReady!, \\
\qquad\quad state := state \oplus \\
\qquad\qquad (\{(target?, nonexistent)\} \cup \{(topReady!, running)\}), \\
\qquad\quad tasks := tasks \setminus \{target?\}] \\
\qquad \wedge\ (st \in TASK \\
\qquad\quad \wedge \neg\ (state \oplus (\{(target?, nonexistent)\} \cup \\
\qquad\qquad\quad \{(topReady!, running)\}))st = state(st) \\
\qquad\quad \Rightarrow (state(st), (state \oplus (\{(target?, nonexistent)\} \cup \\
\qquad\qquad\quad \{(topReady!, running)\}))st) \in transition) \\
\qquad \Rightarrow t \in TASK \\
\qquad\quad \wedge\ state(t) = ready \\
\qquad\quad \wedge \neg\ priority(topReady!) \geq priority(t))
\end{array}
$$

theorem DeleteTaskS_TQ_vc_ref

$\forall\, DeleteTaskS\_TQFSBSig \mid true \bullet \text{pre}\ DeleteTaskS\_TQ$

$$DeleteTask\_TQ \mathrel{\widehat=} DeleteTaskN\_TQ \vee DeleteTaskS\_TQ$$

$$ExecuteRunningTask\_TQ \mathrel{\widehat=} ExtendTaskXi \wedge ExecuteRunningTask\_T$$

$$
\begin{array}{l}
ExecuteRunningTask\_TQFSBSig \mathrel{\widehat=} ExtTaskFSBSig \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ ExecuteRunningTask\_TFSBSig
\end{array}
$$

theorem ExecuteRunningTask_TQ_vc_ref

$\forall\, ExecuteRunningTask\_TQFSBSig \mid true \bullet \text{pre}\ ExecuteRunningTask\_TQ$

SuspendTaskN_TQ

$$
\begin{array}{l}
SuspendTaskN\_T \\
\Delta TaskQueue \\
\hline
running\_task \notin \text{dom}\ release\_snd \cup \text{dom}\ release\_rcv \\
\Xi QueueData \\
wait\_snd' = \{target?\} \lhd wait\_snd \\
wait\_rcv' = \{target?\} \lhd wait\_rcv \\
\Xi QReleasingData
\end{array}
$$

$$SuspendTaskN\_TQFSBSig \mathrel{\widehat=} ExtTaskFSBSig \wedge SuspendTaskN\_TFSBSig$$

theorem SuspendTaskN_TQ_vc_ref

$$\forall\, SuspendTaskN\_TQFSBSig \mid true \bullet \text{pre}\ SuspendTaskN\_TQ$$

$$SuspendTaskS\_TQ \mathrel{\widehat=} ExtendTaskXi \land SuspendTaskS\_T$$

$$SuspendTaskS\_TQFSBSig \mathrel{\widehat=} ExtTaskFSBSig \land SuspendTaskS\_TFSBSig$$

theorem lSuspendTaskS_TQ_Lemma

$$\begin{array}{l}
\forall\, TaskQueue;\ target?, topReady! : TASK \\
\quad \mid running_task \notin \text{dom}\, release_snd \cup \text{dom}\, release_rcv \\
\quad \land\ target? \in tasks \setminus \{idle\} \\
\quad \land\ state(target?) \in \{running\} \\
\quad \land\ state(topReady!) = ready \\
\quad \land\ (\forall\, rtsk : state^{\sim}(\!|\ \{ready\}\ |\!) \bullet priority(topReady!) \geq priority(rtsk)) \\
\quad \bullet \neg\ (TaskQueue[log_context := log_context \oplus \\
\qquad\qquad \{(running_task, phys_context)\}, \\
\qquad\quad phys_context := log_context(topReady!), \\
\qquad\quad running_task := topReady!, \\
\qquad\quad state := state \oplus (\{(running_task, suspended)\} \cup \\
\qquad\qquad \{(topReady!, running)\})] \\
\qquad \land\ (st \in TASK \\
\qquad\quad \land \neg\ (state \oplus (\{(running_task, suspended)\} \cup \\
\qquad\qquad\quad \{(topReady!, running)\}))st = state(st) \\
\qquad\quad \Rightarrow (state(st), (state \oplus (\{(running_task, suspended)\} \cup \\
\qquad\qquad\quad \{(topReady!, running)\}))st) \in transition) \\
\qquad \Rightarrow t \in TASK \\
\qquad\quad \land\ state(t) = ready \\
\qquad\quad \land \neg\ priority(topReady!) \geq priority(t))
\end{array}$$

theorem SuspendTaskS_TQ_vc_ref

$$\forall\, SuspendTaskS\_TQFSBSig \mid true \bullet \text{pre}\ SuspendTaskS\_TQ$$

$$SuspendTaskO\_TQ \mathrel{\widehat=} ExtendTaskXi \land SuspendTaskO\_T$$

$$SuspendTaskO\_TQFSBSig \mathrel{\widehat=} ExtTaskFSBSig \land SuspendTaskO\_TFSBSig$$

theorem SuspendTaskO_TQ_vc_ref

$$\forall\, SuspendTaskO\_TQFSBSig \mid true \bullet \text{pre}\ SuspendTaskO\_TQ$$

$$\begin{array}{l}
SuspendTask_TQ \mathrel{\widehat=} SuspendTaskN_TQ \\
\qquad\qquad\qquad \lor\ SuspendTaskS_TQ \\
\qquad\qquad\qquad \lor\ SuspendTaskO_TQ
\end{array}$$

$ResumeTaskN\_TQ \mathrel{\widehat=} ExtendTaskXi \land ResumeTaskN\_T$

$ResumeTaskN\_TQFSBSig \mathrel{\widehat=} ExtTaskFSBSig \land ResumeTaskN\_TFSBSig$

theorem ResumeTaskN_TQ_vc_ref
$\forall\ ResumeTaskN\_TQFSBSig \mid true \bullet \text{pre}\ ResumeTaskN\_TQ$

$ResumeTaskS\_TQ \mathrel{\widehat=} ExtendTaskXi \land ResumeTaskS\_T$

$ResumeTaskS\_TQFSBSig \mathrel{\widehat=} ExtTaskFSBSig \land ResumeTaskS\_TFSBSig$

theorem ResumeTaskS_TQ_vc_ref
$\forall\ ResumeTaskS\_TQFSBSig \mid true \bullet \text{pre}\ ResumeTaskS\_TQ$

$ResumeTask\_TQ \mathrel{\widehat=} ResumeTaskN\_TQ \lor ResumeTaskS\_TQ$

$ChangeTaskPriorityN\_TQ \mathrel{\widehat=} ExtendTaskXi \land ChangeTaskPriorityN\_T$

$ChangeTaskPriorityN\_TQFSBSig \mathrel{\widehat=} ExtTaskFSBSig \land ChangeTaskPriorityN\_TFSBSig$

theorem ChangeTaskPriorityN_TQ_vc_ref
$\forall\ ChangeTaskPriorityN\_TQFSBSig \mid true \bullet \text{pre}\ ChangeTaskPriorityN\_TQ$

$ChangeTaskPriorityS\_TQ \mathrel{\widehat=} ExtendTaskXi \land ChangeTaskPriorityS\_T$

$ChangeTaskPriorityS\_TQFSBSig \mathrel{\widehat=} ExtTaskFSBSig \land ChangeTaskPriorityS\_TFSBSig$

theorem ChangeTaskPriorityS_TQ_vc_ref
$\forall\ ChangeTaskPriorityS\_TQFSBSig \mid true \bullet \text{pre}\ ChangeTaskPriorityS\_TQ$

$ChangeTaskPriorityD\_TQ \mathrel{\widehat=} ExtendTaskXi \land ChangeTaskPriorityD\_T$

$ChangeTaskPriorityD\_TQFSBSig \mathrel{\widehat=} ExtTaskFSBSig \land ChangeTaskPriorityD\_TFSBSig$

theorem lChangeTaskPriorityD_TQ_Lemma

$$
\begin{array}{l}
\forall\, TaskQueue;\ target?, topReady! : TASK;\ newpri? : \mathbb{N} \\
\quad \mid state(target?) = running \\
\quad \land\ (target? = idle \Rightarrow newpri? = 0) \\
\quad \land\ state(topReady!) = ready \\
\quad \land\ (\forall\, rtsk : state^{\sim}(\!|\ \{ready\}\ |\!) \bullet priority(topReady!) \geq priority(rtsk)) \\
\quad \land\ newpri? < priority(topReady!) \\
\quad \bullet \neg\ (TaskQueue[log\_context := log\_context \oplus \\
\qquad\qquad \{(running\_task, phys\_context)\}, \\
\qquad\quad phys\_context := log\_context(topReady!), \\
\qquad\quad priority := priority \oplus \{(target?, newpri?)\}, \\
\qquad\quad running\_task := topReady!, \\
\qquad\quad state := state \oplus \\
\qquad\qquad (\{(running\_task, ready)\} \cup \{(topReady!, running)\})] \\
\qquad \land\ newpri? < priority(topReady!) \\
\qquad \land\ (st \in TASK \\
\qquad\quad \land \neg\ (state \oplus (\{(running\_task, ready)\} \cup \\
\qquad\qquad\quad \{(topReady!, running)\}))st = state(st) \\
\qquad\qquad \Rightarrow (state(st), (state \oplus (\{(running\_task, ready)\} \cup \\
\qquad\qquad\quad \{(topReady!, running)\}))st) \in transition) \\
\qquad \Rightarrow t \in TASK \\
\qquad\quad \land\ state(t) = ready \\
\qquad\quad \land \neg\ priority(topReady!) \geq priority(t))
\end{array}
$$

theorem ChangeTaskPriorityD_TQ_vc_ref

$$
\begin{array}{l}
\forall\, ChangeTaskPriorityD\_TQFSBSig \mid true \\
\quad \bullet\ \text{pre}\ ChangeTaskPriorityD\_TQ
\end{array}
$$

CreateQueue_TQ

$$
\begin{array}{l}
\Delta\, TaskQueue \\
que? : QUEUE \\
size? : \mathbb{N} \\
\hline
running\_task \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv \\
que? \notin queue \\
size? > 0 \\
\Xi\, Task \\
queue' = queue \cup \{que?\} \\
q\_max' = q\_max \oplus \{(que? \mapsto size?)\} \\
q\_size' = q\_size \oplus \{(que? \mapsto 0)\} \\
\Xi\, WaitingData \\
\Xi\, QReleasingData
\end{array}
$$

CreateQueue_TQFSBSig

$TaskQueue$
$que? : QUEUE$
$size? : \mathbb{N}$

$running\_task \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv$
$que? \notin queue$
$size? > 0$

theorem CreateQueue_TQ_vc_ref
$\forall CreateQueue\_TQFSBSig \mid true \bullet \operatorname{pre} CreateQueue\_TQ$

DeleteQueue_TQ

$\Delta TaskQueue$
$que? : QUEUE$

$running\_task \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv$
$que? \in queue$
$que? \notin \operatorname{ran} wait\_snd \cup \operatorname{ran} wait\_rcv$
$que? \notin \operatorname{ran} release\_snd \cup \operatorname{ran} release\_rcv$
$\Xi Task$
$queue' = queue \setminus \{que?\}$
$q\_max' = \{que?\} \lhd q\_max$
$q\_size' = \{que?\} \lhd q\_size$
$\Xi WaitingData$
$\Xi QReleasingData$

DeleteQueue_TQFSBSig

$TaskQueue$
$que? : QUEUE$

$running\_task \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv$
$que? \in queue$
$que? \notin \operatorname{ran} wait\_snd \cup \operatorname{ran} wait\_rcv$
$que? \notin \operatorname{ran} release\_snd \cup \operatorname{ran} release\_rcv$

theorem DeleteQueue_TQ_vc_ref
$\forall DeleteQueue\_TQFSBSig \mid true \bullet \operatorname{pre} DeleteQueue\_TQ$

QueueSendN_TQ

$\Delta TaskQueue$
$que? : QUEUE$
$topReady! : TASK$

$$
\begin{array}{|l}
\hline
running\_task \notin \operatorname{dom} release\_rcv \\
running\_task \in \operatorname{dom} release\_snd \Rightarrow que? = release\_snd(running\_task) \\
que? \in queue \\
q\_size(que?) < q\_max(que?) \\
que? \notin \operatorname{ran} wait\_rcv \\
\Xi Task \\
queue' = queue \\
q\_max' = q\_max \\
q\_size' = q\_size \oplus \{(que? \mapsto q\_size(que?) + 1)\} \\
\Xi WaitingData \\
release\_snd' = \{running\_task\} \lhd release\_snd \\
release\_rcv' = release\_rcv \\
topReady! = running\_task \\
\hline
\end{array}
$$

QueueSendN_TQFSBSig

$$
\begin{array}{|l}
TaskQueue \\
que? : QUEUE \\
\hline
running\_task \notin \operatorname{dom} release\_rcv \\
running\_task \in \operatorname{dom} release\_snd \Rightarrow que? = release\_snd(running\_task) \\
que? \in queue \\
q\_size(que?) < q\_max(que?) \\
que? \notin \operatorname{ran} wait\_rcv \\
\hline
\end{array}
$$

theorem QueueSendN_TQ_vc_ref

$\forall QueueSendN\_TQFSBSig \mid true \bullet \operatorname{pre} QueueSendN\_TQ$

QueueSendF_TQ

$$
\begin{array}{|l}
\Delta TaskQueue \\
que? : QUEUE \\
topReady! : TASK \\
\hline
running\_task \notin \operatorname{dom} release\_rcv \\
running\_task \in \operatorname{dom} release\_snd \Rightarrow que? = release\_snd(running\_task) \\
que? \in queue \\
q\_size(que?) = q\_max(que?) \\
running\_task \neq idle \\
state(topReady!) = ready \\
\forall t : state^{\sim}(\!| \{ready\} |\!) \bullet priority(topReady!) \geq priority(t) \\
\exists st? : STATE \mid st? = blocked \\
\quad \bullet Reschedule[topReady!/target?, tasks/tasks?, priority/pri?] \\
\Xi QueueData \\
wait\_snd' = wait\_snd \oplus \{(running\_task \mapsto que?)\} \\
\end{array}
$$

$$wait\_rcv' = wait\_rcv$$
$$release\_snd' = \{running\_task\} \lhd release\_snd$$
$$release\_rcv' = release\_rcv$$

QueueSendF_TQFSBSig

$TaskQueue$
$que? : QUEUE$

$$running\_task \notin \mathrm{dom}\, release\_rcv$$
$$running\_task \in \mathrm{dom}\, release\_snd \Rightarrow que? = release\_snd(running\_task)$$
$$que? \in queue$$
$$q\_size(que?) = q\_max(que?)$$
$$running\_task \neq idle$$

theorem lQueueSendF_TQ_Lemma

$$\forall\ TaskQueue;\ topReady! : TASK;\ que? : QUEUE$$
$$\mid running\_task \notin \mathrm{dom}\, release\_rcv$$
$$\wedge\ (running\_task \in \mathrm{dom}\, release\_snd$$
$$\Rightarrow que? = release\_snd(running\_task))$$
$$\wedge\ que? \in queue$$
$$\wedge\ q\_size(que?) = q\_max(que?)$$
$$\wedge\ running\_task \neq idle$$
$$\wedge\ state(topReady!) = ready$$
$$\wedge\ (\forall\ rtsk : state^{\sim}(\!|\ \{ready\}\ |\!) \bullet priority(topReady!) \geq priority(rtsk))$$
$$\bullet\ \neg\ (TaskQueue[log\_context := log\_context \oplus$$
$$\{(running\_task, phys\_context)\},$$
$$phys\_context := log\_context(topReady!),$$
$$release\_snd := \{running\_task\} \lhd release\_snd,$$
$$running\_task := topReady!,$$
$$state := state \oplus$$
$$(\{(running\_task, blocked)\} \cup \{(topReady!, running)\}),$$
$$wait\_snd := wait\_snd \oplus \{(running\_task, que?)\}]$$
$$\wedge\ (t \in TASK \wedge state(t) = ready$$
$$\Rightarrow priority(topReady!) \geq priority(t))$$
$$\Rightarrow st \in TASK$$
$$\wedge\ \neg\ (state \oplus (\{(running\_task, blocked)\} \cup$$
$$\{(topReady!, running)\}))st = state(st)$$
$$\wedge\ \neg\ (state(st), (state \oplus (\{(running\_task, blocked)\} \cup$$
$$\{(topReady!, running)\}))st) \in transition)$$

theorem QueueSendF_TQ_vc_ref

$$\forall\ QueueSendF\_TQFSBSig \mid true \bullet \mathrm{pre}\ QueueSendF\_TQ$$

QueueSendW_TQ

$\Delta TaskQueue$
$que? : QUEUE$
$topReady! : TASK$

$running\_task \notin \mathrm{dom}\, release\_rcv$
$running\_task \in \mathrm{dom}\, release\_snd \Rightarrow que? = release\_snd(running\_task)$
$que? \in queue$
$q\_size(que?) < q\_max(que?)$
$topReady! \in wait\_rcv^{\sim}(\!| \{que?\} |\!)$
$\forall wr : wait\_rcv^{\sim}(\!| \{que?\} |\!) \bullet priority(topReady!) \geq priority(wr)$
$priority(topReady!) \leq priority(running\_task)$
$\Xi TaskData$
$state' = state \oplus \{(topReady! \mapsto ready)\}$
$\Xi ContextData$
$\Xi PrioData$
$queue' = queue$
$q\_max' = q\_max$
$q\_size' = q\_size \oplus \{(que? \mapsto q\_size(que?) + 1)\}$
$wait\_snd' = wait\_snd$
$wait\_rcv' = \{topReady!\} \lhd wait\_rcv$
$release\_snd' = \{running\_task\} \lhd release\_snd$
$release\_rcv' = release\_rcv \oplus \{(topReady! \mapsto que?)\}$

QueueSendW_TQFSBSig

$TaskQueue$
$que? : QUEUE$

$running\_task \notin \mathrm{dom}\, release\_rcv$
$running\_task \in \mathrm{dom}\, release\_snd \Rightarrow que? = release\_snd(running\_task)$
$que? \in queue$
$q\_size(que?) < q\_max(que?)$
$\forall wr : wait\_rcv^{\sim}(\!| \{que?\} |\!) \bullet priority(running\_task) \geq priority(wr)$

theorem lQueueSendW_TQ_Lemma

$\forall TaskQueue;\ que? : QUEUE;\ topReady! : TASK$
$\quad | running\_task \notin \mathrm{dom}\, release\_rcv$
$\quad \wedge (running\_task \in \mathrm{dom}\, release\_snd$
$\quad\quad \Rightarrow que? = release\_snd(running\_task))$
$\quad \wedge que? \in queue$
$\quad \wedge q\_size(que?) < q\_max(que?)$
$\quad \wedge topReady! \in wait\_rcv^{\sim}(\!| \{que?\} |\!)$
$\quad \wedge (\forall wrct : wait\_rcv^{\sim}(\!| \{que?\} |\!)$
$\quad\quad \bullet priority(topReady!) \geq priority(wrct))$
$\quad \wedge priority(running\_task) \geq priority(topReady!)$

$$
\begin{array}{l}
\bullet \neg\ (TaskQueue[q\_size := q\_size \oplus \{(que?, (1 + q\_size(que?)))\}, \\
\qquad release\_rcv := release\_rcv \oplus \{(topReady!, que?)\}, \\
\qquad release\_snd := \{running\_task\} \lhd release\_snd, \\
\qquad state := state \oplus \{(topReady!, ready)\}, \\
\qquad wait\_rcv := \{topReady!\} \lhd wait\_rcv] \\
\quad \land\ priority(topReady!) \leq priority(running\_task) \\
\quad \land\ (st \in TASK \land \neg\ (state \oplus \{(topReady!, ready)\})st = state(st) \\
\qquad \Rightarrow (state(st), (state \oplus \{(topReady!, ready)\})st) \\
\qquad\quad \in transition) \\
\quad \Rightarrow wr \in \mathrm{dom}\ wait\_rcv \\
\qquad \land\ wait\_rcv(wr) = que? \\
\qquad \land \neg\ priority(topReady!) \geq priority(wr))
\end{array}
$$

theorem QueueSendW_TQ_vc_ref

$$\forall\ QueueSendW\_TQFSBSig \mid true \bullet \mathrm{pre}\ QueueSendW\_TQ$$

theorem TaskQueueProperty1

$$\forall\ TaskQueue \bullet \forall\ t : TASK \mid t \in \mathrm{dom}\ wait\_rcv \bullet state(t) = blocked$$

QueueSendWS_TQ

$\Delta\, TaskQueue$
$que? : QUEUE$
$topReady! : TASK$

$running\_task \notin \mathrm{dom}\ release\_rcv$
$running\_task \in \mathrm{dom}\ release\_snd \Rightarrow que? = release\_snd(running\_task)$
$que? \in queue$
$q\_size(que?) < q\_max(que?)$
$topReady! \in wait\_rcv^{\sim}(\!|\ \{que?\}\ |\!)$
$\forall\ wr : wait\_rcv^{\sim}(\!|\ \{que?\}\ |\!) \bullet priority(topReady!) \geq priority(wr)$
$priority(topReady!) > priority(running\_task)$
$\exists\ st? : STATE \mid st? = ready$
$\quad \bullet\ Reschedule[topReady!/target?, tasks/tasks?, priority/pri?]$
$queue' = queue$
$q\_max' = q\_max$
$q\_size' = q\_size \oplus \{(que? \mapsto q\_size(que?) + 1)\}$
$wait\_snd' = wait\_snd$
$wait\_rcv' = \{topReady!\} \lhd wait\_rcv$
$release\_snd' = \{running\_task\} \lhd release\_snd$
$release\_rcv' = release\_rcv \oplus \{(topReady! \mapsto que?)\}$

QueueSendWS_TQFSBSig

$TaskQueue$
$que? : QUEUE$

$$
\begin{array}{l}
\hline
\quad running\_task \notin \mathrm{dom}\, release\_rcv \\
\quad running\_task \in \mathrm{dom}\, release\_snd \Rightarrow que? = release\_snd(running\_task) \\
\quad que? \in queue \\
\quad q\_size(que?) < q\_max(que?) \\
\quad \exists\, topReady! : wait\_rcv^{\sim}(\!|\, \{que?\} \,|\!) \\
\qquad \mid \forall\, wr : wait\_rcv^{\sim}(\!|\, \{que?\} \,|\!) \\
\qquad\quad \bullet\ priority(topReady!) \geq priority(wr) \\
\qquad \bullet\ priority(topReady!) > priority(running\_task) \\
\hline
\end{array}
$$

theorem lQueueSendWS_TQ_Lemma

$$
\begin{array}{l}
\quad \forall\, TaskQueue;\ que? : QUEUE;\ topReady! : TASK \\
\qquad \mid running\_task \notin \mathrm{dom}\, release\_rcv \\
\qquad \wedge\ (running\_task \in \mathrm{dom}\, release\_snd \\
\qquad\quad \Rightarrow que? = release\_snd(running\_task)) \\
\qquad \wedge\ que? \in queue \\
\qquad \wedge\ q\_size(que?) < q\_max(que?) \\
\qquad \wedge\ topReady! \in wait\_rcv^{\sim}(\!|\, \{que?\} \,|\!) \\
\qquad \wedge\ (\forall\, wrct : wait\_rcv^{\sim}(\!|\, \{que?\} \,|\!) \\
\qquad\quad \bullet\ priority(topReady!) \geq priority(wrct)) \\
\qquad \wedge\ priority(topReady!) > priority(running\_task) \\
\qquad \bullet\ \neg\ (TaskQueue[log\_context := log\_context \oplus \\
\qquad\qquad\quad \{(running\_task, phys\_context)\}, \\
\qquad\qquad phys\_context := log\_context(topReady!), \\
\qquad\qquad q\_size := q\_size \oplus \{(wait\_rcv(topReady!), (1+ \\
\qquad\qquad\quad q\_size(wait\_rcv(topReady!))))\}, \\
\qquad\qquad release\_rcv := release\_rcv \oplus \\
\qquad\qquad\quad \{(topReady!, wait\_rcv(topReady!))\}, \\
\qquad\qquad release\_snd := \{running\_task\} \lhd release\_snd, \\
\qquad\qquad running\_task := topReady!, \\
\qquad\qquad state := state \oplus \\
\qquad\qquad\quad (\{(running\_task, ready)\} \cup \{(topReady!, running)\}), \\
\qquad\qquad wait\_rcv := \{topReady!\} \lhd wait\_rcv] \\
\qquad\quad \wedge\ (wr \in \mathrm{dom}\, wait\_rcv \wedge wait\_rcv(wr) = wait\_rcv(topReady!) \\
\qquad\qquad \Rightarrow priority(topReady!) \geq priority(wr)) \\
\qquad\quad \Rightarrow st \in TASK \\
\qquad\qquad \wedge\ \neg\ (state \oplus (\{(running\_task, ready)\} \cup \\
\qquad\qquad\qquad \{(topReady!, running)\}))st = state(st) \\
\qquad\qquad\quad \wedge\ \neg\ (state(st), (state \oplus (\{(running\_task, ready)\} \cup \\
\qquad\qquad\qquad \{(topReady!, running)\}))st) \in transition)
\end{array}
$$

theorem QueueSendWS_TQ_vc_ref

$$
\forall\, QueueSendWS\_TQFSBSig \mid true \bullet \mathrm{pre}\ QueueSendWS\_TQ
$$

$$QueueSend\_TQ \mathrel{\widehat=} QueueSendN\_TQ \lor QueueSendF\_TQ \lor QueueSendW\_TQ \lor QueueSendWS\_TQ$$

QueueReceiveN_TQ

$\Delta TaskQueue$
$que? : QUEUE$
$topReady! : TASK$

$running\_task \notin \operatorname{dom} release\_snd$
$running\_task \in \operatorname{dom} release\_rcv \Rightarrow que? = release\_rcv(running\_task)$
$que? \in queue$
$q\_size(que?) > 0$
$que? \notin \operatorname{ran} wait\_snd$
$\Xi Task$
$queue' = queue$
$q\_max' = q\_max$
$q\_size' = q\_size \oplus \{(que? \mapsto q\_size(que?) - 1)\}$
$\Xi WaitingData$
$release\_snd' = release\_snd$
$release\_rcv' = \{running\_task\} \lhd release\_rcv$
$topReady! = running\_task$

QueueReceiveN_TQFSBSig

$TaskQueue$
$que? : QUEUE$

$running\_task \notin \operatorname{dom} release\_snd$
$running\_task \in \operatorname{dom} release\_rcv \Rightarrow que? = release\_rcv(running\_task)$
$que? \in queue$
$q\_size(que?) > 0$
$que? \notin \operatorname{ran} wait\_snd$

theorem QueueReceiveN_TQ_vc_ref

$\forall QueueReceiveN\_TQFSBSig \mid true \bullet \operatorname{pre} QueueReceiveN\_TQ$

QueueReceiveE_TQ

$\Delta TaskQueue$
$que? : QUEUE$
$topReady! : TASK$

$running\_task \notin \operatorname{dom} release\_snd$
$running\_task \in \operatorname{dom} release\_rcv \Rightarrow que? = release\_rcv(running\_task)$

$que? \in queue$
$q\_size(que?) = 0$
$running\_task \neq idle$
$state(topReady!) = ready$
$\forall\, t : state^{\sim}(\!|\, \{ready\} \,|\!) \bullet priority(topReady!) \geq priority(t)$
$\exists\, st? : STATE \mid st? = blocked$
$\quad \bullet\ Reschedule[topReady!/target?, tasks/tasks?, priority/pri?]$
$\Xi QueueData$
$wait\_snd' = wait\_snd$
$wait\_rcv' = wait\_rcv \oplus \{(running\_task \mapsto que?)\}$
$release\_snd' = release\_snd$
$release\_rcv' = \{running\_task\} \lhd release\_rcv$

QueueReceiveE_TQFSBSig

$TaskQueue$
$que? : QUEUE$

$running\_task \notin \operatorname{dom} release\_snd$
$running\_task \in \operatorname{dom} release\_rcv \Rightarrow que? = release\_rcv(running\_task)$
$que? \in queue$
$q\_size(que?) = 0$
$running\_task \neq idle$

theorem lQueueReceiveE_TQ_Lemma

$\forall\, TaskQueue;\ que? : QUEUE;\ topReady! : TASK$
$\quad \mid running\_task \notin \operatorname{dom} release\_snd$
$\quad \wedge\ (running\_task \in \operatorname{dom} release\_rcv$
$\quad\quad \Rightarrow que? = release\_rcv(running\_task))$
$\quad \wedge\ que? \in queue$
$\quad \wedge\ q\_size(que?) = 0$
$\quad \wedge\ running\_task \neq idle$
$\quad \wedge\ topReady! \in state^{\sim}(\!|\, \{ready\} \,|\!)$
$\quad \wedge\ (\forall\, rtsk : state^{\sim}(\!|\, \{ready\} \,|\!)$
$\quad\quad \bullet\ priority(topReady!) \geq priority(rtsk))$
$\quad \bullet\ \neg\ (TaskQueue[log\_context := log\_context \oplus$
$\quad\quad\quad\quad \{(running\_task, phys\_context)\},$
$\quad\quad\quad phys\_context := log\_context(topReady!),$
$\quad\quad\quad release\_rcv := \{running\_task\} \lhd release\_rcv,$
$\quad\quad\quad running\_task := topReady!,$
$\quad\quad\quad state := state \oplus$
$\quad\quad\quad\quad (\{(running\_task, blocked)\} \cup \{(topReady!, running)\}),$
$\quad\quad\quad wait\_rcv := wait\_rcv \oplus \{(running\_task, que?)\}]$
$\quad\quad \wedge\ (st \in TASK$
$\quad\quad\quad \wedge\ \neg\ (state \oplus (\{(running\_task, blocked)\} \cup$
$\quad\quad\quad\quad\quad \{(topReady!, running)\}))st = state(st)$

$$\Rightarrow (state(st), (state \oplus (\{(running\_task, blocked)\} \cup \{(topReady!, running)\}))st) \in transition)$$
$$\Rightarrow t \in TASK$$
$$\land\ state(t) = ready$$
$$\land\ \neg\ priority(topReady!) \geq priority(t))$$

theorem QueueReceiveE_TQ_vc_ref

$$\forall\ QueueReceiveE\_TQFSBSig \mid true \bullet \text{pre}\ QueueReceiveE\_TQ$$

QueueReceiveW_TQ

$\Delta TaskQueue$
$que? : QUEUE$
$topReady! : TASK$

$running\_task \notin \text{dom}\ release\_snd$
$running\_task \in \text{dom}\ release\_rcv \Rightarrow que? = release\_rcv(running\_task)$
$que? \in queue$
$q\_size(que?) > 0$
$topReady! \in wait\_snd^{\sim}(\!|\ \{que?\}\ |\!)$
$\forall\ ws : wait\_snd^{\sim}(\!|\ \{que?\}\ |\!) \bullet priority(topReady!) \geq priority(ws)$
$priority(topReady!) \leq priority(running\_task)$
$\Xi TaskData$
$state' = state \oplus \{(topReady! \mapsto ready)\}$
$\Xi ContextData$
$\Xi PrioData$
$queue' = queue$
$q\_max' = q\_max$
$q\_size' = q\_size \oplus \{(que? \mapsto q\_size(que?) - 1)\}$
$wait\_snd' = \{topReady!\} \lhd wait\_snd$
$wait\_rcv' = wait\_rcv$
$release\_snd' = release\_snd \oplus \{(topReady! \mapsto que?)\}$
$release\_rcv' = \{running\_task\} \lhd release\_rcv$

QueueReceiveW_TQFSBSig

$TaskQueue$
$que? : QUEUE$

$running\_task \notin \text{dom}\ release\_snd$
$running\_task \in \text{dom}\ release\_rcv \Rightarrow que? = release\_rcv(running\_task)$
$que? \in queue$
$q\_size(que?) > 0$
$\forall\ ws : wait\_snd^{\sim}(\!|\ \{que?\}\ |\!) \bullet priority(running\_task) \geq priority(ws)$

theorem lQueueReceiveW_TQ_Lemma

$$
\begin{array}{l}
\forall\, TaskQueue;\ que? : QUEUE;\ topReady! : TASK \\
\quad \mid running\_task \notin \operatorname{dom} release\_snd \\
\quad \land\ (running\_task \in \operatorname{dom} release\_rcv \\
\qquad \Rightarrow que? = release\_rcv(running\_task)) \\
\quad \land\ que? \in queue \\
\quad \land\ q\_size(que?) > 0 \\
\quad \land\ topReady! \in wait\_snd^{\sim}(\!|\ \{que?\}\ |\!) \\
\quad \land\ (\forall\, wsnt : wait\_snd^{\sim}(\!|\ \{que?\}\ |\!) \\
\qquad \bullet\ priority(topReady!) \geq priority(wsnt)) \\
\quad \land\ priority(running\_task) \geq priority(topReady!) \\
\quad \bullet \neg\ (TaskQueue[q\_size := q\_size \oplus \{(que?, (q\_size(que?) - 1))\}, \\
\qquad\quad release\_rcv := \{running\_task\} \lhd release\_rcv, \\
\qquad\quad release\_snd := release\_snd \oplus \{(topReady!, que?)\}, \\
\qquad\quad state := state \oplus \{(topReady!, ready)\}, \\
\qquad\quad wait\_snd := \{topReady!\} \lhd wait\_snd] \\
\qquad \land\ priority(topReady!) \leq priority(running\_task) \\
\qquad \land\ (st \in TASK \\
\qquad\quad \land \neg\ (state \oplus \{(topReady!, ready)\})st = state(st) \\
\qquad\qquad \Rightarrow (state(st), (state \oplus \{(topReady!, ready)\})st) \\
\qquad\qquad\quad \in transition) \\
\qquad \Rightarrow ws \in \operatorname{dom} wait\_snd \\
\qquad\quad \land\ wait\_snd(ws) = que? \\
\qquad\quad \land \neg\ priority(topReady!) \geq priority(ws))
\end{array}
$$

theorem QueueReceiveW_TQ_vc_ref

$$\forall\, QueueReceiveW\_TQFSBSig \mid true \bullet \text{pre}\ QueueReceiveW\_TQ$$

QueueReceiveWS_TQ

$$
\begin{array}{l}
\Delta\, TaskQueue \\
que? : QUEUE \\
topReady! : TASK \\
\hline
running\_task \notin \operatorname{dom} release\_snd \\
running\_task \in \operatorname{dom} release\_rcv \Rightarrow que? = release\_rcv(running\_task) \\
que? \in queue \\
q\_size(que?) > 0 \\
topReady! \in wait\_snd^{\sim}(\!|\ \{que?\}\ |\!) \\
\forall\, ws : wait\_snd^{\sim}(\!|\ \{que?\}\ |\!) \bullet priority(topReady!) \geq priority(ws) \\
priority(topReady!) > priority(running\_task) \\
\exists\, st? : STATE \mid st? = ready \\
\quad \bullet\ Reschedule[topReady!/target?, tasks/tasks?, priority/pri?] \\
queue' = queue \\
q\_max' = q\_max \\
q\_size' = q\_size \oplus \{(que? \mapsto q\_size(que?) - 1)\}
\end{array}
$$

$$wait\_snd' = \{topReady!\} \lhd wait\_snd$$
$$wait\_rcv' = wait\_rcv$$
$$release\_snd' = release\_snd \oplus \{(topReady! \mapsto que?)\}$$
$$release\_rcv' = \{running\_task\} \lhd release\_rcv$$

QueueReceiveWS_TQFSBSig

$TaskQueue$
$que? : QUEUE$

$$running\_task \notin \mathrm{dom}\, release\_snd$$
$$running\_task \in \mathrm{dom}\, release\_rcv \Rightarrow que? = release\_rcv(running\_task)$$
$$que? \in queue$$
$$q\_size(que?) > 0$$
$$\exists\, topReady! : wait\_snd^{\sim}(\!|\, \{que?\} \,|\!)$$
$$\mid \forall\, ws : wait\_snd^{\sim}(\!|\, \{que?\} \,|\!)$$
$$\bullet\ priority(topReady!) \geq priority(ws)$$
$$\bullet\ priority(topReady!) > priority(running\_task)$$

theorem lQueueReceiveWS_TQ_Lemma

$$\forall\, TaskQueue;\ que? : QUEUE;\ topReady! : TASK$$
$$\mid running\_task \notin \mathrm{dom}\, release\_snd$$
$$\wedge\ (running\_task \in \mathrm{dom}\, release\_rcv$$
$$\Rightarrow que? = release\_rcv(running\_task))$$
$$\wedge\ que? \in queue$$
$$\wedge\ q\_size(que?) > 0$$
$$\wedge\ topReady! \in wait\_snd^{\sim}(\!|\, \{que?\} \,|\!)$$
$$\wedge\ (\forall\, wsnt : wait\_snd^{\sim}(\!|\, \{que?\} \,|\!)$$
$$\bullet\ priority(topReady!) \geq priority(wsnt))$$
$$\wedge\ priority(topReady!) > priority(running\_task)$$
$$\bullet\ \neg\ (TaskQueue[log\_context := log\_context \oplus$$
$$\{(running\_task, phys\_context)\},$$
$$phys\_context := log\_context(topReady!),$$
$$q\_size := q\_size \oplus \{(wait\_snd(topReady!),$$
$$(q\_size(wait\_snd(topReady!)) - 1))\},$$
$$release\_rcv := \{running\_task\} \lhd release\_rcv,$$
$$release\_snd := release\_snd \oplus$$
$$\{(topReady!, wait\_snd(topReady!))\},$$
$$running\_task := topReady!,$$
$$state := state \oplus$$
$$(\{(running\_task, ready)\} \cup \{(topReady!, running)\}),$$
$$wait\_snd := \{topReady!\} \lhd wait\_snd]$$
$$\wedge\ (st \in TASK$$
$$\wedge\ \neg\ (state \oplus (\{(running\_task, ready)\} \cup$$
$$\{(topReady!, running)\}))st = state(st)$$
$$\Rightarrow (state(st), (state \oplus (\{(running\_task, ready)\} \cup$$

$$\{(topReady!, running)\}))st) \in transition)$$
$$\Rightarrow ws \in \mathrm{dom}\, wait\_snd$$
$$\wedge\ wait\_snd(ws) = wait\_snd(topReady!)$$
$$\wedge\ \neg\ priority(topReady!) \geq priority(ws))$$

theorem QueueReceiveWS_TQ_vc_ref

$$\forall\ QueueReceiveWS\_TQFSBSig \mid true \bullet \mathrm{pre}\ QueueReceiveWS\_TQ$$

$$QueueReceive\_TQ \mathrel{\widehat=} QueueReceiveN\_TQ$$
$$\vee\ QueueReceiveE\_TQ$$
$$\vee\ QueueReceiveW\_TQ$$
$$\vee\ QueueReceiveWS\_TQ$$

Appendix F

SPECIFICATION FOR TIME MODEL

$$slice\_delay : \mathbb{N}$$

$\langle\!\langle$ disabled slice_delay_def $\rangle\!\rangle$
$slice\_delay = 1$

Time

$$clock : \mathbb{N}$$
$$delayed\_task : \mathbb{P}\ TASK$$
$$wait\_time : TASK \pfun \mathbb{N}$$
$$time\_slice : \mathbb{N}$$

$$\forall\, t : \mathrm{dom}\ wait\_time \bullet wait\_time(t) \geq clock$$

Init_Time

$$Time'$$

$$clock' = 0$$
$$delayed\_task' = \emptyset$$
$$wait\_time' = \emptyset$$
$$time\_slice' = slice\_delay$$

theorem TimeInit
$\exists\, Time' \bullet Init\_Time$

TaskQueueTime

$$TaskQueue$$
$$Time$$

$$\langle delayed\_task, \mathrm{dom}\ wait\_snd, \mathrm{dom}\ wait\_rcv \rangle \ \textsf{partition}\ \mathrm{dom}\ wait\_time$$
$$delayed\_task \subseteq state^{\sim}(\!|\ \{blocked\}\ |\!)$$

theorem rule domTime
$$\forall\ TaskQueueTime;\ t : TASK \mid t \in \mathrm{dom}\ wait\_time \bullet t \in state^{\sim}(\!|\ \{blocked\}\ |\!)$$

Init_TaskQueueTime
$$TaskQueueTime'$$

$$Init\_TaskQueue$$
$$Init\_Time$$

theorem TaskQueueTimeInit
$$\exists\ TaskQueueTime' \bullet Init\_TaskQueueTime$$

$$\Delta TaskQueueTime \mathrel{\widehat=} TaskQueueTime \land TaskQueueTime' \land \Delta Task$$

ExtendTaskQueueXi
$$\Delta TaskQueueTime$$

$$\Xi Time$$

$$CreateTaskN\_TQT \mathrel{\widehat=} ExtendTaskQueueXi \land CreateTaskN\_TQ$$

$$CreateTaskN\_TQTFSBSig \mathrel{\widehat=} TaskQueueTime \land CreateTaskN\_TQFSBSig$$

theorem CreateTaskN_TQT_vc_ref
$$\forall\ CreateTaskN\_TQTFSBSig \mid true \bullet \mathrm{pre}\ CreateTaskN\_TQT$$

$$CreateTaskS\_TQT \mathrel{\widehat=} ExtendTaskQueueXi \land CreateTaskS\_TQ$$

$$CreateTaskS\_TQTFSBSig \mathrel{\widehat=} TaskQueueTime \land CreateTaskS\_TQFSBSig$$

theorem CreateTaskS_TQT_vc_ref
$$\forall\ CreateTaskS\_TQTFSBSig \mid true \bullet \mathrm{pre}\ CreateTaskS\_TQT$$

$$CreateTask\_TQT \mathrel{\widehat=} CreateTaskN\_TQT \lor CreateTaskS\_TQT$$

$DeleteTaskN\_TQT$

$DeleteTaskN\_TQ$
$\Delta\, TaskQueueTime$

$clock' = clock$
$delayed\_task' = delayed\_task \setminus \{target?\}$
$wait\_time' = \{target?\} \lhd wait\_time$
$time\_slice' = time\_slice$

$DeleteTaskN\_TQTFSBSig \mathrel{\widehat=} TaskQueueTime \land DeleteTaskN\_TQFSBSig$

theorem DeleteTaskN_TQT_vc_ref
$\forall\, DeleteTaskN\_TQTFSBSig \mid true \bullet \text{pre}\ DeleteTaskN\_TQT$

$DeleteTaskS\_TQT \mathrel{\widehat=} ExtendTaskQueueXi \land DeleteTaskS\_TQ$

$DeleteTaskS\_TQTFSBSig \mathrel{\widehat=} TaskQueueTime \land DeleteTaskS\_TQFSBSig$

theorem lDeleteTaskS_TQT_Lemma
$\forall\, TaskQueueTime;\ topReady!, target? : TASK$
$\mid running\_task \notin \text{dom}\ release\_snd \cup \text{dom}\ release\_rcv$
$\land\ target? \in tasks \setminus \{idle\}$
$\land\ state(target?) \in \{running\}$
$\land\ state(topReady!) = ready$
$\land\ (\forall\, rtsk : state^{\sim}(\!|\ \{ready\}\ |\!)$
$\bullet\ priority(topReady!) \geq priority(rtsk))$
$\bullet \neg\ (TaskQueueTime[log\_context := log\_context \oplus$
$\{(target?, bare\_context)\},$
$phys\_context := log\_context(topReady!),$
$running\_task := topReady!,$
$state := state \oplus$
$(\{(target?, nonexistent)\} \cup \{(topReady!, running)\}),$
$tasks := tasks \setminus \{target?\}]$
$\land\ (st \in TASK$
$\land \neg\ (state \oplus (\{(target?, nonexistent)\} \cup$
$\{(topReady!, running)\}))st = state(st)$
$\Rightarrow (state(st), (state \oplus (\{(target?, nonexistent)\} \cup$
$\{(topReady!, running)\}))st) \in transition)$
$\Rightarrow\ t \in TASK$
$\land\ state(t) = ready$
$\land \neg\ priority(topReady!) \geq priority(t))$

theorem DeleteTaskS_TQT_vc_ref
$\forall\, DeleteTaskS\_TQTFSBSig \mid true \bullet \text{pre}\ DeleteTaskS\_TQT$

$DeleteTask\_TQT \mathrel{\widehat=} DeleteTaskN\_TQT \lor DeleteTaskS\_TQT$

$ExecuteRunningTask\_TQT \mathrel{\widehat=} ExtendTaskQueueXi \land ExecuteRunningTask\_TQ$

$ExecuteRunningTask\_TQTFSBSig \mathrel{\widehat=} TaskQueueTime \land ExecuteRunningTask\_TQFSBSig$

theorem ExecuteRunningTask_TQT_vc_ref
$\forall\, ExecuteRunningTask\_TQTFSBSig \mid true \bullet \text{pre}\ ExecuteRunningTask\_TQT$

SuspendTaskN_TQT

$SuspendTaskN\_TQ$
$\Delta TaskQueueTime$

$clock' = clock$
$delayed\_task' = delayed\_task \setminus \{target?\}$
$wait\_time' = \{target?\} \lhd wait\_time$
$time\_slice' = time\_slice$

$SuspendTaskN\_TQTFSBSig \mathrel{\widehat=} TaskQueueTime \land SuspendTaskN\_TQFSBSig$

theorem SuspendTaskN_TQT_vc_ref
$\forall\, SuspendTaskN\_TQTFSBSig \mid true \bullet \text{pre}\ SuspendTaskN\_TQT$

$SuspendTaskS\_TQT \mathrel{\widehat=} ExtendTaskQueueXi \land SuspendTaskS\_TQ$

$SuspendTaskS\_TQTFSBSig \mathrel{\widehat=} TaskQueueTime \land SuspendTaskS\_TQFSBSig$

theorem lSuspendTaskS_TQT_Lemma
$\forall\, TaskQueueTime;\ target?, topReady! : TASK$
$\mid running\_task \notin \text{dom}\ release\_snd \cup \text{dom}\ release\_rcv$
$\land\ target? \in tasks \setminus \{idle\}$
$\land\ state(target?) \in \{running\}$
$\land\ state(topReady!) = ready$

$$
\begin{array}{l}
\quad \wedge\ (\forall\, rtsk : state^{\sim}(\!|\ \{ready\}\ |\!) \\
\qquad \bullet\ priority(topReady!) \geq priority(rtsk)) \\
\bullet\ \neg\ (TaskQueueTime[log\_context := log\_context \oplus \\
\qquad\qquad \{(running\_task, phys\_context)\}, \\
\qquad\quad phys\_context := log\_context(topReady!), \\
\qquad\quad running\_task := topReady!, \\
\qquad\quad state := state \oplus (\{(running\_task, suspended)\} \cup \\
\qquad\qquad \{(topReady!, running)\})] \\
\qquad \wedge\ (st \in TASK \\
\qquad\quad \wedge\ \neg\ (state \oplus (\{(running\_task, suspended)\} \cup \\
\qquad\qquad\quad \{(topReady!, running)\}))st = state(st) \\
\qquad\quad \Rightarrow (state(st), (state \oplus (\{(running\_task, suspended)\} \cup \\
\qquad\qquad\quad \{(topReady!, running)\}))st) \in transition) \\
\qquad \Rightarrow t \in TASK \\
\qquad\quad \wedge\ state(t) = ready \\
\qquad\quad \wedge\ \neg\ priority(topReady!) \geq priority(t))
\end{array}
$$

theorem SuspendTaskS_TQT_vc_ref

$\forall\, SuspendTaskS\_TQTFSBSig \mid true \bullet \text{pre}\ SuspendTaskS\_TQT$

$$SuspendTaskO\_TQT \mathrel{\widehat=} ExtendTaskQueueXi \wedge SuspendTaskO\_TQ$$

$$
\begin{array}{l}
SuspendTaskO\_TQTFSBSig \mathrel{\widehat=} TaskQueueTime \\
\qquad\qquad\qquad\qquad \wedge\ SuspendTaskO\_TQFSBSig
\end{array}
$$

theorem SuspendTaskO_TQT_vc_ref

$\forall\, SuspendTaskO\_TQTFSBSig \mid true \bullet \text{pre}\ SuspendTaskO\_TQT$

$$
\begin{array}{l}
SuspendTask\_TQT \mathrel{\widehat=} SuspendTaskN\_TQT \\
\qquad\qquad\qquad\quad \vee\ SuspendTaskS\_TQT \\
\qquad\qquad\qquad\quad \vee\ SuspendTaskO\_TQT
\end{array}
$$

$$ResumeTaskN\_TQT \mathrel{\widehat=} ExtendTaskQueueXi \wedge ResumeTaskN\_TQ$$

$$
\begin{array}{l}
ResumeTaskN\_TQTFSBSig \mathrel{\widehat=} TaskQueueTime \\
\qquad\qquad\qquad\qquad \wedge\ ResumeTaskN\_TQFSBSig
\end{array}
$$

theorem ResumeTaskN_TQT_vc_ref

$\forall\, ResumeTaskN\_TQTFSBSig \mid true \bullet \text{pre}\ ResumeTaskN\_TQT$

$$ResumeTaskS\_TQT \mathrel{\widehat=} ExtendTaskQueueXi \wedge ResumeTaskS\_TQ$$

$$ResumeTaskS\_TQTFSBSig \mathrel{\widehat=} TaskQueueTime \\ \land\ ResumeTaskS\_TQFSBSig$$

theorem ResumeTaskS_TQT_vc_ref
$\forall\ ResumeTaskS\_TQTFSBSig \mid true \bullet \text{pre}\ ResumeTaskS\_TQT$

$$ResumeTask\_TQT \mathrel{\widehat=} ResumeTaskN\_TQT \lor ResumeTaskS\_TQT$$

$$ChangeTaskPriorityN\_TQT \mathrel{\widehat=} ExtendTaskQueueXi \\ \land\ ChangeTaskPriorityN\_TQ$$

$$ChangeTaskPriorityN\_TQTFSBSig \mathrel{\widehat=} TaskQueueTime \\ \land\ ChangeTaskPriorityN\_TQFSBSig$$

theorem ChangeTaskPriorityN_TQT_vc_ref
$\forall\ ChangeTaskPriorityN\_TQTFSBSig \mid true$
$\bullet\ \text{pre}\ ChangeTaskPriorityN\_TQT$

$$ChangeTaskPriorityS\_TQT \mathrel{\widehat=} ExtendTaskQueueXi \\ \land\ ChangeTaskPriorityS\_TQ$$

$$ChangeTaskPriorityS\_TQTFSBSig \mathrel{\widehat=} TaskQueueTime \\ \land\ ChangeTaskPriorityS\_TQFSBSig$$

theorem ChangeTaskPriorityS_TQT_vc_ref
$\forall\ ChangeTaskPriorityS\_TQTFSBSig \mid true$
$\bullet\ \text{pre}\ ChangeTaskPriorityS\_TQT$

$$ChangeTaskPriorityD\_TQT \mathrel{\widehat=} ExtendTaskQueueXi \\ \land\ ChangeTaskPriorityD\_TQ$$

$$ChangeTaskPriorityD\_TQTFSBSig \mathrel{\widehat=} TaskQueueTime \\ \land\ ChangeTaskPriorityD\_TQFSBSig$$

theorem lChangeTaskPriorityD_TQT_Lemma
$\forall\ TaskQueueTime;\ target?, topReady! : TASK;\ newpri? : \mathbb{N}$
$\mid state(target?) = running$
$\land\ (target? = idle \Rightarrow newpri? = 0)$
$\land\ state(topReady!) = ready$
$\land\ (\forall\ rtsk : state^{\sim}(\!|\ \{ready\}\ |\!)$

$$
\begin{array}{l}
\quad \bullet\ priority(topReady!) \geq priority(rtsk)) \\
\wedge\ newpri? < priority(topReady!) \\
\bullet\ \neg\ (TaskQueueTime[log\_context := log\_context \oplus \\
\qquad \{(running\_task, phys\_context)\}, \\
\qquad phys\_context := log\_context(topReady!), \\
\qquad priority := priority \oplus \{(target?, newpri?)\}, \\
\qquad running\_task := topReady!, \\
\qquad state := state \oplus \\
\qquad\quad (\{(running\_task, ready)\} \cup \{(topReady!, running)\})] \\
\quad \wedge\ newpri? < priority(topReady!) \\
\quad \wedge\ (st \in TASK \\
\qquad \wedge\ \neg\ (state \oplus (\{(running\_task, ready)\} \cup \\
\qquad\quad \{(topReady!, running)\}))st = state(st) \\
\qquad\quad \Rightarrow (state(st), (state \oplus (\{(running\_task, ready)\} \cup \\
\qquad\quad\quad \{(topReady!, running)\}))st) \in transition) \\
\quad \Rightarrow t \in TASK \\
\qquad \wedge\ state(t) = ready \\
\qquad \wedge\ \neg\ priority(topReady!) \geq priority(t))
\end{array}
$$

theorem ChangeTaskPriorityD_TQT_vc_ref

$\forall\ ChangeTaskPriorityD\_TQTFSBSig \mid true$

• pre $ChangeTaskPriorityD\_TQT$

$$CreateQueue\_TQT \mathrel{\widehat=} ExtendTaskQueueXi \wedge CreateQueue\_TQ$$

$$CreateQueue\_TQTFSBSig \mathrel{\widehat=} TaskQueueTime \wedge CreateQueue\_TQFSBSig$$

theorem CreateQueue_TQT_vc_ref

$\forall\ CreateQueue\_TQTFSBSig \mid true \bullet$ pre $CreateQueue\_TQT$

$$DeleteQueue\_TQT \mathrel{\widehat=} ExtendTaskQueueXi \wedge DeleteQueue\_TQ$$

$$DeleteQueue\_TQTFSBSig \mathrel{\widehat=} TaskQueueTime \wedge DeleteQueue\_TQFSBSig$$

theorem DeleteQueue_TQT_vc_ref

$\forall\ DeleteQueue\_TQTFSBSig \mid true \bullet$ pre $DeleteQueue\_TQT$

$$QueueSendN\_TQT \mathrel{\widehat=} ExtendTaskQueueXi \wedge QueueSendN\_TQ$$

$$QueueSendN\_TQTFSBSig \mathrel{\widehat=} TaskQueueTime \wedge QueueSendN\_TQFSBSig$$

theorem QueueSendN_TQT_vc_ref

$\forall\, QueueSendN\_TQTFSBSig \mid true \bullet \text{pre}\ QueueSendN\_TQT$

QueueSendF_TQT

$\Delta TaskQueueTime$

$QueueSendF\_TQ$

$wtime? : \mathbb{N}$

$wtime? > clock$

$clock' = clock$

$delayed\_task' = delayed\_task$

$wait\_time' = wait\_time \oplus \{(running\_task \mapsto wtime?)\}$

$time\_slice' = time\_slice$

QueueSendF_TQTFSBSig

$TaskQueueTime$

$QueueSendF\_TQFSBSig$

$wtime? : \mathbb{N}$

$wtime? > clock$

theorem lQueueSendF_TQT_Lemma

$$
\begin{array}{l}
\forall\, TaskQueueTime;\ topReady! : TASK;\ que? : QUEUE;\ wtime? : \mathbb{N} \\
\quad \mid running\_task \notin \operatorname{dom} release\_rcv \\
\quad \land\ (running\_task \in \operatorname{dom} release\_snd \\
\qquad \Rightarrow que? = release\_snd(running\_task)) \\
\quad \land\ que? \in queue \\
\quad \land\ q\_size(que?) = q\_max(que?) \\
\quad \land\ running\_task \neq idle \\
\quad \land\ state(topReady!) = ready \\
\quad \land\ (\forall\, rtsk : state^{\sim}(\!|\ \{ready\}\ |\!) \\
\qquad \bullet\ priority(topReady!) \geq priority(rtsk)) \\
\quad \land\ wtime? > clock \\
\quad \bullet \neg\ (TaskQueueTime[log\_context := log\_context \oplus \\
\qquad\qquad \{(running\_task, phys\_context)\}, \\
\qquad\quad phys\_context := log\_context(topReady!), \\
\qquad\quad release\_snd := \{running\_task\} \lhd release\_snd, \\
\qquad\quad running\_task := topReady!, \\
\qquad\quad state := state \oplus \\
\qquad\qquad (\{(running\_task, blocked)\} \cup \{(topReady!, running)\}), \\
\qquad\quad wait\_time := wait\_time \oplus \{(running\_task, wtime?)\}, \\
\qquad\quad wait\_snd := wait\_snd \oplus \{(running\_task, que?)\}] \\
\qquad \land\ (st \in TASK \\
\qquad\quad \land \neg\ (state \oplus (\{(running\_task, blocked)\} \cup
\end{array}
$$

$$\{(topReady!, running)\}))st = state(st)$$
$$\Rightarrow (state(st), (state \oplus (\{(running\_task, blocked)\} \cup$$
$$\{(topReady!, running)\}))st) \in transition)$$
$$\Rightarrow t \in TASK$$
$$\wedge\ state(t) = ready$$
$$\wedge\ \neg\ priority(topReady!) \geq priority(t))$$

theorem QueueSendF_TQT_vc_ref

$$\forall\ QueueSendF\_TQTFSBSig \mid true \bullet \text{pre}\ QueueSendF\_TQT$$

QueueSendW_TQT

$$\Delta TaskQueueTime$$
$$QueueSendW\_TQ$$

$$clock' = clock$$
$$delayed\_task' = delayed\_task$$
$$wait\_time' = \{topReady!\} \lhd wait\_time$$
$$time\_slice' = time\_slice$$

$$QueueSendW\_TQTFSBSig \mathrel{\widehat=} TaskQueueTime$$
$$\wedge\ QueueSendW\_TQFSBSig$$

theorem lQueueSendW_TQT_Lemma

$$\forall\ TaskQueueTime;\ topReady! : TASK;\ que? : QUEUE$$
$$\mid running\_task \notin \text{dom}\ release\_rcv$$
$$\wedge\ (running\_task \in \text{dom}\ release\_snd$$
$$\Rightarrow que? = release\_snd(running\_task))$$
$$\wedge\ que? \in queue$$
$$\wedge\ q\_size(que?) < q\_max(que?)$$
$$\wedge\ topReady! \in wait\_rcv^{\sim}(\!|\ \{que?\}\ |\!)$$
$$\wedge\ (\forall\ wrct : wait\_rcv^{\sim}(\!|\ \{que?\}\ |\!)$$
$$\bullet\ priority(topReady!) \geq priority(wrct))$$
$$\wedge\ priority(running\_task) \geq priority(topReady!)$$
$$\bullet \neg\ (TaskQueueTime[q\_size := q\_size \oplus \{(que?, (1+$$
$$q\_size(que?)))\},$$
$$release\_rcv := release\_rcv \oplus \{(topReady!, que?)\},$$
$$release\_snd := \{running\_task\} \lhd release\_snd,$$
$$state := state \oplus \{(topReady!, ready)\},$$
$$wait\_time := \{topReady!\} \lhd wait\_time,$$
$$wait\_rcv := \{topReady!\} \lhd wait\_rcv]$$
$$\wedge\ priority(topReady!) \leq priority(running\_task)$$
$$\wedge\ (st \in TASK$$
$$\wedge\ \neg\ (state \oplus \{(topReady!, ready)\})st = state(st)$$
$$\Rightarrow (state(st), (state\oplus$$

$$\{(topReady!, ready)\})st) \in transition)$$
$$\Rightarrow wr \in \mathrm{dom}\, wait\_rcv$$
$$\wedge\ wait\_rcv(wr) = que?$$
$$\wedge\ \neg\ priority(topReady!) \geq priority(wr))$$

theorem QueueSendW_TQT_vc_ref

$$\forall\ QueueSendW\_TQTFSBSig \mid true \bullet \mathrm{pre}\ QueueSendW\_TQT$$

QueueSendWS_TQT

$$\Delta TaskQueueTime$$
$$QueueSendWS\_TQ$$

$$clock' = clock$$
$$delayed\_task' = delayed\_task$$
$$wait\_time' = \{topReady!\} \lhd wait\_time$$
$$time\_slice' = time\_slice$$

$$QueueSendWS\_TQTFSBSig \mathrel{\widehat=} TaskQueueTime \wedge QueueSendWS\_TQFSBSig$$

theorem lQueueSendWS_TQT_Lemma

$$\forall\ TaskQueueTime;\ topReady! : TASK;\ que? : QUEUE$$
$$\mid running\_task \notin \mathrm{dom}\, release\_rcv$$
$$\wedge\ (running\_task \in \mathrm{dom}\, release\_snd$$
$$\Rightarrow que? = release\_snd(running\_task))$$
$$\wedge\ que? \in queue$$
$$\wedge\ q\_size(que?) < q\_max(que?)$$
$$\wedge\ topReady! \in wait\_rcv^{\sim}(\!|\ \{que?\}\ |\!)$$
$$\wedge\ (\forall\ wrct : wait\_rcv^{\sim}(\!|\ \{que?\}\ |\!)$$
$$\bullet\ priority(topReady!) \geq priority(wrct))$$
$$\wedge\ priority(topReady!) > priority(running\_task)$$
$$\bullet \neg\ (TaskQueueTime[log\_context := log\_context \oplus$$
$$\{(running\_task, phys\_context)\},$$
$$phys\_context := log\_context(topReady!),$$
$$q\_size := q\_size \oplus \{(wait\_rcv(topReady!), (1+$$
$$q\_size(wait\_rcv(topReady!)))\},$$
$$release\_rcv := release\_rcv \oplus$$
$$\{(topReady!, wait\_rcv(topReady!))\},$$
$$release\_snd := \{running\_task\} \lhd release\_snd,$$
$$running\_task := topReady!,$$
$$state := state \oplus$$
$$(\{(running\_task, ready)\} \cup \{(topReady!, running)\}),$$
$$wait\_time := \{topReady!\} \lhd wait\_time,$$
$$wait\_rcv := \{topReady!\} \lhd wait\_rcv]$$

$$
\begin{array}{l}
\quad \wedge\ (st \in TASK \\
\qquad \wedge\ \neg\ (state \oplus (\{(running\_task, ready)\} \cup \\
\qquad\qquad \{(topReady!, running)\}))st = state(st) \\
\qquad\quad \Rightarrow (state(st), (state \oplus (\{(running\_task, ready)\} \cup \\
\qquad\qquad \{(topReady!, running)\}))st) \in transition) \\
\Rightarrow\ wr \in \mathrm{dom}\ wait\_rcv \\
\qquad \wedge\ wait\_rcv(wr) = wait\_rcv(topReady!) \\
\qquad \wedge\ \neg\ priority(topReady!) \geq priority(wr))
\end{array}
$$

theorem QueueSendWS_TQT_vc_ref

$$\forall\ QueueSendWS\_TQTFSBSig \mid true \bullet \mathrm{pre}\ QueueSendWS\_TQT$$

$$
\begin{array}{l}
QueueSend\_TQT \mathrel{\widehat=} QueueSendN\_TQT \\
\qquad\qquad\qquad \vee\ QueueSendF\_TQT \\
\qquad\qquad\qquad \vee\ QueueSendW\_TQT \\
\qquad\qquad\qquad \vee\ QueueSendWS\_TQT
\end{array}
$$

$$QueueReceiveN\_TQT \mathrel{\widehat=} ExtendTaskQueueXi \wedge QueueReceiveN\_TQ$$

$$
\begin{array}{l}
QueueReceiveN\_TQTFSBSig \mathrel{\widehat=} TaskQueueTime \\
\qquad\qquad\qquad\qquad\qquad \wedge\ QueueReceiveN\_TQFSBSig
\end{array}
$$

theorem QueueReceiveN_TQT_vc_ref

$$\forall\ QueueReceiveN\_TQTFSBSig \mid true \bullet \mathrm{pre}\ QueueReceiveN\_TQT$$

QueueReceiveE_TQT

$$
\begin{array}{l}
\Delta TaskQueueTime \\
QueueReceiveE\_TQ \\
wtime? : \mathbb{N} \\
\hline
wtime? > clock \\
clock' = clock \\
delayed\_task' = delayed\_task \\
wait\_time' = wait\_time \oplus \{(running\_task \mapsto wtime?)\} \\
time\_slice' = time\_slice
\end{array}
$$

QueueReceiveE_TQTFSBSig

$$
\begin{array}{l}
TaskQueueTime \\
QueueReceiveE\_TQFSBSig \\
wtime? : \mathbb{N} \\
\hline
wtime? > clock
\end{array}
$$

theorem lQueueReceiveE_TQT_Lemma

$$
\begin{array}{l}
\forall\, TaskQueueTime;\ que? : QUEUE;\ topReady! : TASK;\ wtime? : \mathbb{N} \\
\quad \mid running\_task \notin \operatorname{dom} release\_snd \\
\quad \land\ (running\_task \in \operatorname{dom} release\_rcv \\
\qquad \Rightarrow que? = release\_rcv(running\_task)) \\
\quad \land\ que? \in queue \\
\quad \land\ q\_size(que?) = 0 \\
\quad \land\ running\_task \neq idle \\
\quad \land\ topReady! \in state^{\sim}(\!|\ \{ready\}\ |\!) \\
\quad \land\ (\forall\, rtsk : state^{\sim}(\!|\ \{ready\}\ |\!) \\
\qquad \bullet\ priority(topReady!) \geq priority(rtsk)) \\
\quad \land\ wtime? > clock \\
\quad \bullet\ \neg\ (TaskQueueTime[log\_context := log\_context \oplus \\
\qquad\qquad \{(running\_task, phys\_context)\}, \\
\qquad\quad phys\_context := log\_context(topReady!), \\
\qquad\quad release\_rcv := \{running\_task\} \lhd release\_rcv, \\
\qquad\quad running\_task := topReady!, \\
\qquad\quad state := state \oplus \\
\qquad\qquad (\{(running\_task, blocked)\} \cup \{(topReady!, running)\}), \\
\qquad\quad wait\_time := wait\_time \oplus \{(running\_task, wtime?)\}, \\
\qquad\quad wait\_rcv := wait\_rcv \oplus \{(running\_task, que?)\}] \\
\qquad \land\ (st \in TASK \\
\qquad\quad \land\ \neg\ (state \oplus (\{(running\_task, blocked)\} \cup \\
\qquad\qquad\quad \{(topReady!, running)\}))st = state(st) \\
\qquad\quad\ \Rightarrow (state(st), (state \oplus (\{(running\_task, blocked)\} \cup \\
\qquad\qquad\quad \{(topReady!, running)\}))st) \in transition) \\
\qquad \Rightarrow t \in TASK \\
\qquad\quad \land\ state(t) = ready \\
\qquad\quad \land\ \neg\ priority(topReady!) \geq priority(t))
\end{array}
$$

theorem QueueReceiveE_TQT_vc_ref

$$\forall\, QueueReceiveE\_TQTFSBSig \mid true \bullet \text{pre}\ QueueReceiveE\_TQT$$

QueueReceiveW_TQT

$$
\begin{array}{l}
\Delta\, TaskQueueTime \\
QueueReceiveW\_TQ \\
\hline
clock' = clock \\
delayed\_task' = delayed\_task \\
wait\_time' = \{topReady!\} \lhd wait\_time \\
time\_slice' = time\_slice
\end{array}
$$

$$
\begin{array}{l}
QueueReceiveW\_TQTFSBSig \mathrel{\widehat=} TaskQueueTime \\
\qquad\qquad\qquad\qquad\qquad \land\ QueueReceiveW\_TQFSBSig
\end{array}
$$

theorem lQueueReceiveW_TQT_Lemma

$$
\begin{array}{l}
\forall\, TaskQueueTime;\ que? : QUEUE;\ topReady! : TASK \\
\quad \mid running\_task \notin \operatorname{dom} release\_snd \\
\quad \land\ (running\_task \in \operatorname{dom} release\_rcv \\
\qquad \Rightarrow que? = release\_rcv(running\_task)) \\
\quad \land\ que? \in queue \\
\quad \land\ q\_size(que?) > 0 \\
\quad \land\ topReady! \in wait\_snd^{\sim}(\!|\ \{que?\}\ |\!) \\
\quad \land\ (\forall\, wsnt : wait\_snd^{\sim}(\!|\ \{que?\}\ |\!) \\
\qquad \bullet\ priority(topReady!) \geq priority(wsnt)) \\
\quad \land\ priority(running\_task) \geq priority(topReady!) \\
\quad \bullet\ \neg\ (TaskQueueTime[q\_size := q\_size \oplus \\
\qquad\qquad \{(que?, (q\_size(que?) - 1))\}, \\
\qquad\quad release\_rcv := \{running\_task\} \lhd release\_rcv, \\
\qquad\quad release\_snd := release\_snd \oplus \{(topReady!, que?)\}, \\
\qquad\quad state := state \oplus \{(topReady!, ready)\}, \\
\qquad\quad wait\_time := \{topReady!\} \lhd wait\_time, \\
\qquad\quad wait\_snd := \{topReady!\} \lhd wait\_snd] \\
\qquad \land\ priority(topReady!) \leq priority(running\_task) \\
\qquad \land\ (st \in TASK \\
\qquad\quad \land\ \neg\ (state \oplus \{(topReady!, ready)\})st = state(st) \\
\qquad\qquad \Rightarrow (state(st), (state \oplus \\
\qquad\qquad\quad \{(topReady!, ready)\})st) \in transition) \\
\qquad \Rightarrow ws \in \operatorname{dom} wait\_snd \\
\qquad\quad \land\ wait\_snd(ws) = que? \\
\qquad\quad \land\ \neg\ priority(topReady!) \geq priority(ws))
\end{array}
$$

theorem QueueReceiveW_TQT_vc_ref

$$\forall\, QueueReceiveW\_TQTFSBSig \mid true \bullet \text{pre}\ QueueReceiveW\_TQT$$

QueueReceiveWS_TQT

$$
\begin{array}{l}
\Delta\, TaskQueueTime \\
QueueReceiveWS\_TQ \\
\hline
clock' = clock \\
delayed\_task' = delayed\_task \\
wait\_time' = \{topReady!\} \lhd wait\_time \\
time\_slice' = time\_slice
\end{array}
$$

$$
\begin{array}{l}
QueueReceiveWS\_TQTFSBSig \mathrel{\widehat=} TaskQueueTime \\
\qquad\qquad\qquad\qquad\qquad \land\ QueueReceiveWS\_TQFSBSig
\end{array}
$$

theorem lQueueReceiveWS_TQT_Lemma

$$\forall\, TaskQueueTime;\ que? : QUEUE;\ topReady! : TASK$$

$$
\begin{array}{l}
\mid running\_task \notin \mathrm{dom}\, release\_snd \\
\land\ (running\_task \in \mathrm{dom}\, release\_rcv \\
\qquad \Rightarrow que? = release\_rcv(running\_task)) \\
\land\ que? \in queue \\
\land\ q\_size(que?) > 0 \\
\land\ topReady! \in wait\_snd^{\sim}(\!|\ \{que?\}\ |\!) \\
\land\ (\forall\, wsnt : wait\_snd^{\sim}(\!|\ \{que?\}\ |\!) \\
\qquad \bullet\ priority(topReady!) \geq priority(wsnt)) \\
\land\ priority(topReady!) > priority(running\_task) \\
\bullet\ \neg\ (TaskQueueTime[log\_context := log\_context \oplus \\
\qquad\qquad \{(running\_task, phys\_context)\}, \\
\qquad phys\_context := log\_context(topReady!), \\
\qquad q\_size := q\_size \oplus \{(wait\_snd(topReady!), \\
\qquad\qquad (q\_size(wait\_snd(topReady!)) - 1))\}, \\
\qquad release\_rcv := \{running\_task\} \lhd release\_rcv, \\
\qquad release\_snd := release\_snd \oplus \\
\qquad\qquad \{(topReady!, wait\_snd(topReady!))\}, \\
\qquad running\_task := topReady!, \\
\qquad state := state \oplus \\
\qquad\qquad (\{(running\_task, ready)\} \cup \{(topReady!, running)\}), \\
\qquad wait\_time := \{topReady!\} \lhd wait\_time, \\
\qquad wait\_snd := \{topReady!\} \lhd wait\_snd] \\
\qquad \land\ (st \in TASK \\
\qquad\qquad \land\ \neg\ (state \oplus (\{(running\_task, ready)\} \cup \\
\qquad\qquad\qquad \{(topReady!, running)\}))st = state(st) \\
\qquad\qquad\qquad \Rightarrow (state(st), (state \oplus (\{(running\_task, ready)\} \cup \\
\qquad\qquad\qquad\qquad \{(topReady!, running)\}))st) \in transition) \\
\qquad \Rightarrow ws \in \mathrm{dom}\, wait\_snd \\
\qquad\qquad \land\ wait\_snd(ws) = wait\_snd(topReady!) \\
\qquad\qquad \land\ \neg\ priority(topReady!) \geq priority(ws))
\end{array}
$$

theorem QueueReceiveWS_TQT_vc_ref

$$\forall\, QueueReceiveWS\_TQTFSBSig \mid true \bullet \mathrm{pre}\ QueueReceiveWS\_TQT$$

$$
\begin{array}{l}
QueueReceive\_TQT \mathrel{\widehat=} QueueReceiveN\_TQT \\
\qquad\qquad\qquad\qquad \lor\ QueueReceiveE\_TQT \\
\qquad\qquad\qquad\qquad \lor\ QueueReceiveW\_TQT \\
\qquad\qquad\qquad\qquad \lor\ QueueReceiveWS\_TQT
\end{array}
$$

DelayUntil_TQT

$$
\begin{array}{l}
\Delta TaskQueueTime \\
wtime? : \mathbb{N} \\
topReady! : TASK \\
\hline
running\_task \notin \mathrm{dom}\, release\_snd \cup \mathrm{dom}\, release\_rcv
\end{array}
$$

$state(topReady!) = ready$
$running\_task \neq idle$
$\forall\, t : TASK \mid state(t) = ready \bullet priority(topReady!) \geq priority(t)$
$wtime? > clock$
$\exists\, st? : STATE \mid st? = blocked$
$\quad \bullet\ Reschedule[topReady!/target?, tasks/tasks?, priority/pri?]$
$\Xi Queue$
$clock' = clock$
$delayed\_task' = delayed\_task \cup \{running\_task\}$
$wait\_time' = wait\_time \oplus \{(running\_task \mapsto wtime?)\}$
$time\_slice' = time\_slice$

DelayUntil_TQTFSBSig

$TaskQueueTime$
$wtime? : \mathbb{N}$

$running\_task \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv$
$running\_task \neq idle$
$wtime? > clock$

theorem lDelayUntil_TQT_Lemma

$\forall\, TaskQueueTime;\ wtime? : \mathbb{N};\ topReady! : TASK$
$\quad \mid running\_task \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv$
$\quad \land\ running\_task \neq idle$
$\quad \land\ state(topReady!) = ready$
$\quad \land\ wtime? > clock$
$\quad \land\ (\forall\, rtsk : state^{\sim}(\!|\ \{ready\}\ |\!)$
$\quad\quad \bullet\ priority(topReady!) \geq priority(rtsk))$
$\quad \bullet \neg\ (TaskQueueTime[delayed\_task := delayed\_task \cup$
$\quad\quad\quad \{running\_task\},$
$\quad\quad log\_context := log\_context \oplus$
$\quad\quad\quad \{(running\_task, phys\_context)\},$
$\quad\quad phys\_context := log\_context(topReady!),$
$\quad\quad running\_task := topReady!,$
$\quad\quad state := state \oplus$
$\quad\quad\quad (\{(running\_task, blocked)\} \cup \{(topReady!, running)\}),$
$\quad\quad wait\_time := wait\_time \oplus \{(running\_task, wtime?)\}]$
$\quad\quad \land\ (st \in TASK$
$\quad\quad\quad \land \neg\ (state \oplus (\{(running\_task, blocked)\} \cup$
$\quad\quad\quad\quad \{(topReady!, running)\}))st = state(st)$
$\quad\quad\quad\quad \Rightarrow (state(st), (state \oplus (\{(running\_task, blocked)\} \cup$
$\quad\quad\quad\quad\quad \{(topReady!, running)\}))st) \in transition)$
$\quad\quad \Rightarrow t \in TASK$
$\quad\quad\quad \land\ state(t) = ready$
$\quad\quad\quad \land \neg\ priority(topReady!) \geq priority(t))$

theorem DelayUntil_TQT_vc_ref

$\forall\, DelayUntil\_TQTFSBSig \mid true \bullet \text{pre}\ DelayUntil\_TQT$

CheckDelayedTaskN_TQT

$\Delta TaskQueueTime$

$topWaiting! : TASK$

$running\_task \notin \text{dom}\ release\_snd \cup \text{dom}\ release\_rcv$

$topWaiting! \in \text{dom}\ wait\_time$

$\forall\, wt : \text{dom}\ wait\_time \bullet wait\_time(topWaiting!) \leq wait\_time(wt)$

$\forall\, wt : \text{dom}\ wait\_time \mid wait\_time(wt) = wait\_time(topWaiting!)$
$\quad \bullet\ priority(topWaiting!) \geq priority(wt)$

$priority(topWaiting!) \leq priority(running\_task)$

$\Xi TaskData$

$state' = state \oplus \{(topWaiting! \mapsto ready)\}$

$\Xi ContextData$

$\Xi PrioData$

$\Xi QueueData$

$wait\_snd' = \{topWaiting!\} \lhd wait\_snd$

$wait\_rcv' = \{topWaiting!\} \lhd wait\_rcv$

$\Xi QReleasingData$

$clock' = wait\_time(topWaiting!)$

$delayed\_task' = delayed\_task \setminus \{topWaiting!\}$

$wait\_time' = \{topWaiting!\} \lhd wait\_time$

$time\_slice' = time\_slice$

CheckDelayedTaskN_TQTFSBSig

$TaskQueueTime$

$running\_task \notin \text{dom}\ release\_snd \cup \text{dom}\ release\_rcv$

$\exists\, topWaiting! : \text{dom}\ wait\_time$
$\quad \bullet\ (\forall\, wt : \text{dom}\ wait\_time \bullet wait\_time(topWaiting!) \leq wait\_time(wt))$
$\quad \wedge\ (\forall\, wt : \text{dom}\ wait\_time \mid wait\_time(wt) = wait\_time(topWaiting!)$
$\qquad \bullet\ priority(topWaiting!) \geq priority(wt))$
$\quad \wedge\ priority(topWaiting!) \leq priority(running\_task)$

theorem lCheckDelayedTaskN_TQT_Lemma

$\forall\, TaskQueueTime;\ topWaiting! : TASK$
$\quad \mid running\_task \notin \text{dom}\ release\_snd \cup \text{dom}\ release\_rcv$
$\quad \wedge\ topWaiting! \in \text{dom}\ wait\_time$
$\quad \wedge\ (\forall\, dtk : \text{dom}\ wait\_time$
$\qquad \bullet\ wait\_time(topWaiting!) \leq wait\_time(dtk))$
$\quad \wedge\ (\forall\, detk : \text{dom}\ wait\_time$
$\qquad \mid wait\_time(detk) = wait\_time(topWaiting!)$

$$
\begin{array}{l}
\quad \bullet\ priority(topWaiting!) \geq priority(detk)) \\
\land\ priority(topWaiting!) \leq priority(running\_task) \\
\bullet\ \neg\ (TaskQueueTime[clock := wait\_time(topWaiting!), \\
\qquad delayed\_task := delayed\_task \setminus \{topWaiting!\}, \\
\qquad state := state \oplus \{(topWaiting!, ready)\}, \\
\qquad wait\_time := \{topWaiting!\} \lhd wait\_time, \\
\qquad wait\_rcv := \{topWaiting!\} \lhd wait\_rcv, \\
\qquad wait\_snd := \{topWaiting!\} \lhd wait\_snd] \\
\quad \land\ (st \in TASK \\
\qquad \land \neg\ (state \oplus \{(topWaiting!, ready)\})st = state(st) \\
\qquad\quad \Rightarrow (state(st), (state \oplus \\
\qquad\qquad \{(topWaiting!, ready)\})st) \in transition) \\
\quad \land\ (wt \in \mathrm{dom}\ wait\_time \\
\qquad \Rightarrow wait\_time(topWaiting!) \leq wait\_time(wt)) \\
\quad \Rightarrow wt\_\_0 \in \mathrm{dom}\ wait\_time \\
\qquad \land\ wait\_time(wt\_\_0) = wait\_time(topWaiting!) \\
\qquad \land \neg\ priority(topWaiting!) \geq priority(wt\_\_0))
\end{array}
$$

theorem CheckDelayedTaskN_TQT_vc_ref

$$
\begin{array}{l}
\forall\ CheckDelayedTaskN\_TQTFSBSig \mid true \\
\quad \bullet\ \mathrm{pre}\ CheckDelayedTaskN\_TQT
\end{array}
$$

CheckDelayedTaskS_TQT

$$
\begin{array}{l}
\Delta TaskQueueTime \\
topWaiting! : TASK \\
\hline
running\_task \notin \mathrm{dom}\ release\_snd \cup \mathrm{dom}\ release\_rcv \\
topWaiting! \in \mathrm{dom}\ wait\_time \\
\forall\ wt : \mathrm{dom}\ wait\_time \bullet wait\_time(topWaiting!) \leq wait\_time(wt) \\
\forall\ wt : \mathrm{dom}\ wait\_time \mid wait\_time(wt) = wait\_time(topWaiting!) \\
\quad \bullet\ priority(topWaiting!) \geq priority(wt) \\
priority(topWaiting!) > priority(running\_task) \\
\exists\ st? : STATE \mid st? = ready \\
\quad \bullet\ Reschedule[topWaiting!/target?, tasks/tasks?, priority/pri?] \\
\Xi QueueData \\
wait\_snd' = \{topWaiting!\} \lhd wait\_snd \\
wait\_rcv' = \{topWaiting!\} \lhd wait\_rcv \\
\Xi QReleasingData \\
clock' = wait\_time(topWaiting!) \\
delayed\_task' = delayed\_task \setminus \{topWaiting!\} \\
wait\_time' = \{topWaiting!\} \lhd wait\_time \\
time\_slice' = time\_slice
\end{array}
$$

CheckDelayedTaskS_TQTFSBSig

$TaskQueueTime$

$running\_task \notin \mathrm{dom}\ release\_snd \cup \mathrm{dom}\ release\_rcv$

$\exists\ topWaiting! : \mathrm{dom}\ wait\_time$

$\quad \bullet\ (\forall\ wt : \mathrm{dom}\ wait\_time \bullet wait\_time(topWaiting!) \leq wait\_time(wt))$

$\quad \wedge\ (\forall\ wt : \mathrm{dom}\ wait\_time \mid wait\_time(wt) = wait\_time(topWaiting!)$

$\quad\quad \bullet\ priority(topWaiting!) \geq priority(wt))$

$\quad \wedge\ priority(topWaiting!) > priority(running\_task)$

theorem lCheckDelayedTaskS_TQT_Lemma

$\forall\ TaskQueueTime;\ topWaiting! : TASK$

$\quad \mid running\_task \notin \mathrm{dom}\ release\_snd \cup \mathrm{dom}\ release\_rcv$

$\quad \wedge\ topWaiting! \in \mathrm{dom}\ wait\_time$

$\quad \wedge\ (\forall\ dtk : \mathrm{dom}\ wait\_time$

$\quad\quad \bullet\ wait\_time(topWaiting!) \leq wait\_time(dtk))$

$\quad \wedge\ (\forall\ detk : \mathrm{dom}\ wait\_time$

$\quad\quad \mid wait\_time(detk) = wait\_time(topWaiting!)$

$\quad\quad\quad \bullet\ priority(topWaiting!) \geq priority(detk))$

$\quad \wedge\ priority(topWaiting!) > priority(running\_task)$

$\quad \bullet\ \neg\ (TaskQueueTime[clock := wait\_time(topWaiting!),$

$\quad\quad delayed\_task := delayed\_task \setminus \{topWaiting!\},$

$\quad\quad log\_context := log\_context \oplus$

$\quad\quad\quad \{(running\_task, phys\_context)\},$

$\quad\quad phys\_context := log\_context(topWaiting!),$

$\quad\quad running\_task := topWaiting!,$

$\quad\quad state := state \oplus$

$\quad\quad\quad (\{(running\_task, ready)\} \cup \{(topWaiting!, running)\}),$

$\quad\quad wait\_time := \{topWaiting!\} \lhd wait\_time,$

$\quad\quad wait\_rcv := \{topWaiting!\} \lhd wait\_rcv,$

$\quad\quad wait\_snd := \{topWaiting!\} \lhd wait\_snd]$

$\quad\quad \wedge\ (st \in TASK$

$\quad\quad\quad \wedge\ \neg\ (state \oplus (\{(running\_task, ready)\} \cup$

$\quad\quad\quad\quad \{(topWaiting!, running)\}))st = state(st)$

$\quad\quad\quad\quad \Rightarrow (state(st), (state \oplus (\{(running\_task, ready)\} \cup$

$\quad\quad\quad\quad\quad \{(topWaiting!, running)\}))st) \in transition)$

$\quad\quad \wedge\ (wt \in \mathrm{dom}\ wait\_time$

$\quad\quad\quad \Rightarrow wait\_time(topWaiting!) \leq wait\_time(wt))$

$\quad\quad \Rightarrow wt\_\_0 \in \mathrm{dom}\ wait\_time$

$\quad\quad\quad \wedge\ wait\_time(wt\_\_0) = wait\_time(topWaiting!)$

$\quad\quad\quad \wedge\ \neg\ priority(topWaiting!) \geq priority(wt\_\_0))$

theorem CheckDelayedTaskS_TQT_vc_ref

$\forall\ CheckDelayedTaskS\_TQTFSBSig \mid true$

$\quad \bullet$ pre $CheckDelayedTaskS\_TQT$

TimeSlicing_TQT

$\Delta TaskQueueTime$
$topReady! : TASK$

$running\_task \notin \mathrm{dom}\, release\_snd \cup \mathrm{dom}\, release\_rcv$
$state(topReady!) = ready$
$priority(topReady!) = priority(running\_task)$
$\forall t : \mathrm{dom}\, wait\_time \bullet time\_slice \leq wait\_time(t)$
$\exists st? : STATE \mid st? = ready$
$\quad \bullet Reschedule[topReady!/target?, tasks/tasks?, priority/pri?]$
$\Xi Queue$
$clock' = clock$
$delayed\_task' = delayed\_task$
$wait\_time' = wait\_time$
$time\_slice' = time\_slice + slice\_delay$

TimeSlicing_TQTFSBSig

$TaskQueueTime$

$running\_task \notin \mathrm{dom}\, release\_snd \cup \mathrm{dom}\, release\_rcv$
$\forall t : \mathrm{dom}\, wait\_time \bullet time\_slice \leq wait\_time(t)$
$\exists topReady! : state^{\sim}(\!|\, \{ready\} \,|\!)$
$\quad \bullet priority(topReady!) = priority(running\_task)$

theorem lTimeSlicing_TQT_Lemma

$\forall TaskQueueTime;\ topReady! : TASK$
$\quad \mid running\_task \notin \mathrm{dom}\, release\_snd \cup \mathrm{dom}\, release\_rcv$
$\quad \land\ state(topReady!) = ready$
$\quad \land\ priority(topReady!) = priority(running\_task)$
$\quad \land\ (\forall ts : \mathrm{dom}\, wait\_time \bullet time\_slice \leq wait\_time(ts))$
$\quad \bullet \neg\ (TaskQueueTime[log\_context := log\_context \oplus$
$\qquad \{(running\_task, phys\_context)\},$
$\qquad phys\_context := log\_context(topReady!),$
$\qquad running\_task := topReady!,$
$\qquad state := state \oplus$
$\qquad\quad (\{(running\_task, ready)\} \cup \{(topReady!, running)\}),$
$\qquad time\_slice := time\_slice + slice\_delay]$
$\quad\ \Rightarrow st \in TASK$
$\qquad \land \neg\ (state \oplus (\{(running\_task, ready)\} \cup$
$\qquad\quad \{(topReady!, running)\}))st = state(st)$
$\qquad \land \neg\ (state(st), (state \oplus (\{(running\_task, ready)\} \cup$
$\qquad\quad \{(topReady!, running)\}))st) \in transition)$

theorem TimeSlicing_TQT_vc_ref

$\forall\, TimeSlicing\_TQTFSBSig \mid true \bullet \text{pre}\ TimeSlicing\_TQT$

NoSlicing_TQT

$\Delta\, TaskQueueTime$

$running\_task \notin \text{dom}\ release\_snd \cup \text{dom}\ release\_rcv$

$\forall\, t : \text{dom}\ wait\_time \bullet time\_slice \leq wait\_time(t)$

$\forall\, t : state^{\sim}(\!|\ \{ready\}\ |\!) \bullet priority(t) < priority(running\_task)$

$\Xi\, Task$

$\Xi\, Queue$

$clock' = clock$

$delayed\_task' = delayed\_task$

$wait\_time' = wait\_time$

$time\_slice' = time\_slice + slice\_delay$

NoSlicing_TQTFSBSig

$TaskQueueTime$

$running\_task \notin \text{dom}\ release\_snd \cup \text{dom}\ release\_rcv$

$\forall\, t : \text{dom}\ wait\_time \bullet time\_slice \leq wait\_time(t)$

$\forall\, t : state^{\sim}(\!|\ \{ready\}\ |\!) \bullet priority(t) < priority(running\_task)$

theorem NoSlicing_TQT_vc_ref

$\forall\, NoSlicing\_TQTFSBSig \mid true \bullet \text{pre}\ NoSlicing\_TQT$

Appendix G

SPECIFICATION FOR MUTEX MODEL

MutexData

$$semaphore : \mathbb{P}\, QUEUE$$
$$mutex : \mathbb{P}\, QUEUE$$
$$mutex\_holder : QUEUE \pfun TASK$$
$$mutex\_recursive : QUEUE \pfun \mathbb{N}$$

$$mutex \cap semaphore = \emptyset$$
$$\mathrm{dom}\, mutex\_recursive = mutex$$
$$\forall\, m : mutex \bullet m \notin \mathrm{dom}\, mutex\_holder \Leftrightarrow mutex\_recursive(m) = 0$$

Init_MutexData

$$MutexData'$$

$$semaphore' = \emptyset$$
$$mutex' = \emptyset$$
$$mutex\_holder' = \emptyset$$
$$mutex\_recursive' = \emptyset$$

theorem MutexDataInit

$$\exists\, MutexData' \bullet Init\_MutexData$$

OriginalPrioData

$$base\_priority : TASK \pfun \mathbb{N}$$

Init_OriginalPrioData

$OriginalPrioData'$

$base\_priority' = \emptyset$

theorem OriginalPrioDataInit

$\exists\, OriginalPrioData' \bullet Init\_OriginalPrioData$

MReleasingData

$release\_mutex : TASK \pfun QUEUE$

Init_MReleasingData

$MReleasingData'$

$release\_mutex' = \emptyset$

theorem MReleasingDataInit

$\exists\, MReleasingData' \bullet Init\_MReleasingData$

Mutex

$MutexData$
$OriginalPrioData$
$MReleasingData$

$\mathrm{dom}\ base\_priority = \mathrm{ran}\ mutex\_holder$
$\mathrm{ran}\ release\_mutex \subseteq mutex$

Init_Mutex

$Mutex'$

$Init\_MutexData$
$Init\_OriginalPrioData$
$Init\_MReleasingData$

theorem MutexInit

$\exists\, Mutex' \bullet Init\_Mutex$

TaskQueueTimeMutex

$TaskQueueTime$
$Mutex$

$$
\begin{array}{l}
semaphore \subseteq queue \\
\forall\, s : semaphore \bullet q\_max(s) = 1 \\
mutex \subseteq queue \\
\forall\, m : mutex \bullet q\_max(m) = 1 \\
\mathrm{dom}\, mutex\_holder = \{m : mutex \mid q\_size(m) = 0\} \\
\mathrm{ran}\, mutex\_holder \subseteq tasks \\
\forall\, mh : \mathrm{ran}\, mutex\_holder \bullet priority(mh) \geq base\_priority(mh) \\
\forall\, ms : mutex \cup semaphore \bullet ms \notin \mathrm{ran}\, wait\_snd \cup \mathrm{ran}\, release\_snd \\
release\_mutex \subseteq release\_rcv
\end{array}
$$

$$
\Delta TaskQueueTimeMutex \mathrel{\widehat=} TaskQueueTimeMutex \\
\qquad \land\ TaskQueueTimeMutex' \\
\qquad \land\ \Delta Task
$$

Init_TaskQueueTimeMutex

$$
\begin{array}{l}
TaskQueueTimeMutex' \\
\hline
Init\_TaskQueueTime \\
Init\_Mutex
\end{array}
$$

theorem TaskQueueTimeMutexInit
$\exists\, TaskQueueTimeMutex' \bullet Init\_TaskQueueTimeMutex$

ExtendTQTXi

$$
\begin{array}{l}
\Delta TaskQueueTimeMutex \\
\hline
\Xi Mutex
\end{array}
$$

$$
CreateTaskN\_TQTM \mathrel{\widehat=} ExtendTQTXi \land CreateTaskN\_TQT
$$

$$
CreateTaskN\_TQTMFSBSig \mathrel{\widehat=} TaskQueueTimeMutex \\
\qquad \land\ CreateTaskN\_TQTFSBSig
$$

theorem CreateTaskN_TQTM_vc_ref
$\forall\, CreateTaskN\_TQTMFSBSig \mid true \bullet \mathrm{pre}\ CreateTaskN\_TQTM$

$$
CreateTaskS\_TQTM \mathrel{\widehat=} ExtendTQTXi \land CreateTaskS\_TQT
$$

$$
CreateTaskS\_TQTMFSBSig \mathrel{\widehat=} TaskQueueTimeMutex \\
\qquad \land\ CreateTaskS\_TQTFSBSig
$$

theorem CreateTaskS_TQTM_vc_ref
$\forall\, CreateTaskS\_TQTMFSBSig \mid true \bullet \text{pre}\ CreateTaskS\_TQTM$

$CreateTask\_TQTM \mathrel{\widehat=} CreateTaskN\_TQTM \lor CreateTaskS\_TQTM$

DeleteTaskN_TQTM

$\Delta\, TaskQueueTimeMutex$
$DeleteTaskN\_TQT$

$target? \notin \text{ran}\ mutex\_holder$
$\Xi MutexData$
$\Xi OriginalPrioData$
$release\_mutex' = \{target?\} \lhd release\_mutex$

DeleteTaskN_TQTMFSBSig

$TaskQueueTimeMutex$
$DeleteTaskN\_TQTFSBSig$

$target? \notin \text{ran}\ mutex\_holder$

theorem DeleteTaskN_TQTM_vc_ref
$\forall\, DeleteTaskN\_TQTMFSBSig \mid true \bullet \text{pre}\ DeleteTaskN\_TQTM$

DeleteTaskS_TQTM

$\Delta\, TaskQueueTimeMutex$
$DeleteTaskS\_TQT$

$target? \notin \text{ran}\ mutex\_holder$
$\Xi Mutex$

DeleteTaskS_TQTMFSBSig

$TaskQueueTimeMutex$
$DeleteTaskS\_TQTFSBSig$

$target? \notin \text{ran}\ mutex\_holder$

theorem lDeleteTaskS_TQTM_Lemma
$\forall\, TaskQueueTimeMutex;\ topReady!, target? : TASK$
$\mid running\_task \notin \text{dom}\ release\_snd \cup \text{dom}\ release\_rcv$
$\land\ target? \in tasks \setminus \{idle\}$
$\land\ state(target?) \in \{running\}$

$$
\begin{array}{l}
\quad \wedge\ state(topReady!) = ready \\
\quad \wedge\ (\forall\, rtsk : state^{\sim}(\!|\ \{ready\}\ |\!) \\
\qquad \bullet\ priority(topReady!) \geq priority(rtsk)) \\
\quad \wedge\ target? \notin \operatorname{ran} mutex\_holder \\
\quad \bullet\ \neg\ (TaskQueueTimeMutex[log\_context := log\_context \oplus \\
\qquad\qquad \{(target?, bare\_context)\}, \\
\qquad\quad phys\_context := log\_context(topReady!), \\
\qquad\quad running\_task := topReady!, \\
\qquad\quad state := state \oplus \\
\qquad\qquad (\{(target?, nonexistent)\} \cup \{(topReady!, running)\}), \\
\qquad\quad tasks := tasks \setminus \{target?\}] \\
\qquad \wedge\ (st \in TASK \\
\qquad\quad \wedge\ \neg\ (state \oplus (\{(target?, nonexistent)\} \cup \\
\qquad\qquad\quad \{(topReady!, running)\}))st = state(st) \\
\qquad\quad \Rightarrow (state(st), (state \oplus (\{(target?, nonexistent)\} \cup \\
\qquad\qquad\quad \{(topReady!, running)\}))st) \in transition) \\
\qquad \Rightarrow t \in TASK \\
\qquad\quad \wedge\ state(t) = ready \\
\qquad\quad \wedge\ \neg\ priority(topReady!) \geq priority(t))
\end{array}
$$

theorem DeleteTaskS_TQTM_vc_ref

$\forall\, DeleteTaskS\_TQTMFSBSig \mid true \bullet \text{pre}\ DeleteTaskS\_TQTM$

$$DeleteTask\_TQTM \mathrel{\widehat=} DeleteTaskN\_TQTM \vee DeleteTaskS\_TQTM$$

$$
\begin{array}{l}
ExecuteRunningTask\_TQTM \mathrel{\widehat=} ExtendTQTXi \\
\qquad\qquad\qquad\qquad \wedge\ ExecuteRunningTask\_TQT
\end{array}
$$

$$
\begin{array}{l}
ExecuteRunningTask\_TQTMFSBSig \mathrel{\widehat=} TaskQueueTimeMutex \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ ExecuteRunningTask\_TQTFSBSig
\end{array}
$$

theorem ExecuteRunningTask_TQTM_vc_ref

$\forall\, ExecuteRunningTask\_TQTMFSBSig \mid true$
$\quad \bullet \text{pre}\ ExecuteRunningTask\_TQTM$

$$SuspendTaskN\_TQTM \mathrel{\widehat=} ExtendTQTXi \wedge SuspendTaskN\_TQT$$

$$
\begin{array}{l}
SuspendTaskN\_TQTMFSBSig \mathrel{\widehat=} TaskQueueTimeMutex \\
\qquad\qquad\qquad\qquad\qquad \wedge\ SuspendTaskN\_TQTFSBSig
\end{array}
$$

theorem SuspendTaskN_TQTM_vc_ref

$\forall\, SuspendTaskN\_TQTMFSBSig \mid true \bullet \text{pre}\ SuspendTaskN\_TQTM$

$$SuspendTaskS\_TQTM \mathrel{\widehat=} ExtendTQTXi \land SuspendTaskS\_TQT$$

$$SuspendTaskS\_TQTMFSBSig \mathrel{\widehat=} TaskQueueTimeMutex \land SuspendTaskS\_TQTFSBSig$$

theorem lSuspendTaskS_TQTM_Lemma

$$\begin{array}{l}
\forall\, TaskQueueTimeMutex;\ target?, topReady! : TASK \\
\quad \mid running_task \notin \operatorname{dom} release_snd \cup \operatorname{dom} release_rcv \\
\quad \land\ target? \in tasks \setminus \{idle\} \\
\quad \land\ state(target?) \in \{running\} \\
\quad \land\ state(topReady!) = ready \\
\quad \land\ (\forall\, rtsk : state^{\sim}(\!|\ \{ready\}\ |\!) \\
\qquad \bullet\ priority(topReady!) \geq priority(rtsk)) \\
\quad \bullet\ \neg\ (TaskQueueTimeMutex[log_context := log_context \oplus \\
\qquad \{(running_task, phys_context)\}, \\
\qquad phys_context := log_context(topReady!), \\
\qquad running_task := topReady!, \\
\qquad state := state \oplus (\{(running_task, suspended)\} \cup \\
\qquad \{(topReady!, running)\})] \\
\qquad \land\ (st \in TASK \\
\qquad \land\ \neg\ (state \oplus (\{(running_task, suspended)\} \cup \\
\qquad \{(topReady!, running)\}))st = state(st) \\
\qquad \Rightarrow (state(st), (state \oplus (\{(running_task, suspended)\} \cup \\
\qquad \{(topReady!, running)\}))st) \in transition) \\
\qquad \Rightarrow t \in TASK \\
\qquad \land\ state(t) = ready \\
\qquad \land\ \neg\ priority(topReady!) \geq priority(t))
\end{array}$$

theorem SuspendTaskS_TQTM_vc_ref

$$\forall\, SuspendTaskS\_TQTMFSBSig \mid true \bullet \text{pre}\ SuspendTaskS\_TQTM$$

$$SuspendTaskO\_TQTM \mathrel{\widehat=} ExtendTQTXi \land SuspendTaskO\_TQT$$

$$SuspendTaskO\_TQTMFSBSig \mathrel{\widehat=} TaskQueueTimeMutex \land SuspendTaskO\_TQTFSBSig$$

theorem SuspendTaskO_TQTM_vc_ref

$$\forall\, SuspendTaskO\_TQTMFSBSig \mid true \bullet \text{pre}\ SuspendTaskO\_TQTM$$

$$\begin{array}{l}
SuspendTask_TQTM \mathrel{\widehat=} SuspendTaskN_TQTM \\
\qquad\qquad \lor\ SuspendTaskS_TQTM \\
\qquad\qquad \lor\ SuspendTaskO_TQTM
\end{array}$$

$ResumeTaskN\_TQTM \mathrel{\widehat=} ExtendTQTXi \land ResumeTaskN\_TQT$

$ResumeTaskN\_TQTMFSBSig \mathrel{\widehat=} TaskQueueTimeMutex \land ResumeTaskN\_TQTFSBSig$

theorem ResumeTaskN_TQTM_vc_ref
$\forall ResumeTaskN\_TQTMFSBSig \mid true \bullet \text{pre } ResumeTaskN\_TQTM$

$ResumeTaskS\_TQTM \mathrel{\widehat=} ExtendTQTXi \land ResumeTaskS\_TQT$

$ResumeTaskS\_TQTMFSBSig \mathrel{\widehat=} TaskQueueTimeMutex \land ResumeTaskS\_TQTFSBSig$

theorem ResumeTaskS_TQTM_vc_ref
$\forall ResumeTaskS\_TQTMFSBSig \mid true \bullet \text{pre } ResumeTaskS\_TQTM$

$ResumeTask\_TQTM \mathrel{\widehat=} ResumeTaskN\_TQTM \lor ResumeTaskS\_TQTM$

ChangeTaskPriorityNNotHolder_TQTM

$\Delta TaskQueueTimeMutex$
$ChangeTaskPriorityN\_TQT$

$target? \notin \text{dom } base\_priority$
$\Xi Mutex$

ChangeTaskPriorityNNotHolder_TQTMFSBSig

$ChangeTaskPriorityN\_TQTFSBSig$
$TaskQueueTimeMutex$

$target? \notin \text{dom } base\_priority$

theorem ChangeTaskPriorityNNotHolder_TQTM_vc_ref
$\forall ChangeTaskPriorityNNotHolder\_TQTMFSBSig \mid true \bullet \text{pre } ChangeTaskPriorityNNotHolder\_TQTM$

ChangeTaskPrioritySNotHolder_TQTM

$ChangeTaskPriorityS\_TQT$
$\Delta TaskQueueTimeMutex$

$target? \notin \text{dom } base\_priority$
$\Xi Mutex$

ChangeTaskPrioritySNotHolder_TQTMFSBSig

ChangeTaskPriorityS_TQTFSBSig
TaskQueueTimeMutex

$target? \notin \mathrm{dom}\ base\_priority$

theorem ChangeTaskPrioritySNotHolder_TQTM_vc_ref
$\forall\ ChangeTaskPrioritySNotHolder\_TQTMFSBSig \mid true$
• pre *ChangeTaskPrioritySNotHolder_TQTM*

ChangeTaskPriorityDNotHolder_TQTM

$\Delta\ TaskQueueTimeMutex$
ChangeTaskPriorityD_TQT

$target? \notin \mathrm{dom}\ base\_priority$
$\Xi Mutex$

ChangeTaskPriorityDNotHolder_TQTMFSBSig

TaskQueueTimeMutex
ChangeTaskPriorityD_TQTFSBSig

$target? \notin \mathrm{dom}\ base\_priority$

theorem lChangeTaskPriorityDNotHolder_TQTM_Lemma
$\forall\ TaskQueueTimeMutex;\ target?, topReady! : TASK;\ newpri? : \mathbb{N}$
$\mid state(target?) = running$
$\land\ (target? = idle \Rightarrow newpri? = 0)$
$\land\ state(topReady!) = ready$
$\land\ (\forall\ rtsk : state^{\sim}(\!|\ \{ready\}\ |\!)$
• $priority(topReady!) \geq priority(rtsk))$
$\land\ newpri? < priority(topReady!)$
$\land\ target? \notin \mathrm{dom}\ base\_priority$
• $\neg\ (TaskQueueTimeMutex[log\_context := log\_context \oplus$
$\{(running\_task, phys\_context)\},$
$phys\_context := log\_context(topReady!),$
$priority := priority \oplus \{(target?, newpri?)\},$
$running\_task := topReady!,$
$state := state \oplus$
$(\{(running\_task, ready)\} \cup \{(topReady!, running)\})]$
$\land\ newpri? < priority(topReady!)$
$\land\ (st \in TASK$

$$\begin{array}{l}
\quad \land \neg\ (state \oplus (\{(running_task, ready)\} \cup \\
\qquad \{(topReady!, running)\}))st = state(st) \\
\qquad \Rightarrow (state(st), (state \oplus (\{(running_task, ready)\} \cup \\
\qquad \{(topReady!, running)\}))st) \in transition) \\
\Rightarrow t \in TASK \\
\quad \land state(t) = ready \\
\quad \land \neg\ priority(topReady!) \geq priority(t))
\end{array}$$

theorem ChangeTaskPriorityDNotHolder_TQTM_vc_ref

$\forall\ ChangeTaskPriorityDNotHolder\_TQTMFSBSig \mid true$

• pre $ChangeTaskPriorityDNotHolder\_TQTM$

ChangeTaskPriorityNNotInherited_TQTM

$ChangeTaskPriorityN\_TQT$

$\Delta TaskQueueTimeMutex$

$target? \in \mathrm{dom}\ base\_priority$

$base\_priority(target?) = priority(target?)$

$\Xi MutexData$

$base\_priority' = base\_priority \oplus \{(target? \mapsto newpri?)\}$

$\Xi MReleasingData$

ChangeTaskPriorityNNotInherited_TQTMFSBSig

$ChangeTaskPriorityN\_TQTFSBSig$

$TaskQueueTimeMutex$

$target? \in \mathrm{dom}\ base\_priority$

$base\_priority(target?) = priority(target?)$

theorem ChangeTaskPriorityNNotInherited_TQTM_vc_ref

$\forall\ ChangeTaskPriorityNNotInherited\_TQTMFSBSig \mid true$

• pre $ChangeTaskPriorityNNotInherited\_TQTM$

ChangeTaskPrioritySNotInherited_TQTM

$ChangeTaskPriorityS\_TQT$

$\Delta TaskQueueTimeMutex$

$target? \in \mathrm{dom}\ base\_priority$

$base\_priority(target?) = priority(target?)$

$\Xi MutexData$

$base\_priority' = base\_priority \oplus \{(target? \mapsto newpri?)\}$

$\Xi MReleasingData$

ChangeTaskPrioritySNotInherited_TQTMFSBSig

$ChangeTaskPriorityS\_TQTFSBSig$
$TaskQueueTimeMutex$

$target? \in \mathrm{dom}\ base\_priority$
$base\_priority(target?) = priority(target?)$

theorem ChangeTaskPrioritySNotInherited_TQTM_vc_ref
$\forall\ ChangeTaskPrioritySNotInherited\_TQTMFSBSig \mid true$
• pre $ChangeTaskPrioritySNotInherited\_TQTM$

ChangeTaskPriorityDNotInherited_TQTM

$\Delta\, TaskQueueTimeMutex$
$ChangeTaskPriorityD\_TQT$

$target? \in \mathrm{dom}\ base\_priority$
$base\_priority(target?) = priority(target?)$
$\Xi MutexData$
$base\_priority' = base\_priority \oplus \{(target? \mapsto newpri?)\}$
$\Xi MReleasingData$

ChangeTaskPriorityDNotInherited_TQTMFSBSig

$TaskQueueTimeMutex$
$ChangeTaskPriorityD\_TQTFSBSig$

$target? \in \mathrm{dom}\ base\_priority$
$base\_priority(target?) = priority(target?)$

theorem lChangeTaskPriorityDNotInherited_TQTM_Lemma
$\forall\ TaskQueueTimeMutex;\ target?, topReady! : TASK;\ newpri? : \mathbb{N}$
$\mid state(target?) = running$
$\land\ (target? = idle \Rightarrow newpri? = 0)$
$\land\ state(topReady!) = ready$
$\land\ (\forall\, rtsk : state^{\sim}(\!|\ \{ready\}\ |\!)$
• $priority(topReady!) \geq priority(rtsk))$
$\land\ newpri? < priority(topReady!)$
$\land\ target? \in \mathrm{dom}\ base\_priority$
$\land\ base\_priority(target?) = priority(target?)$
• $\neg\ (TaskQueueTimeMutex[base\_priority := base\_priority \oplus$
$\{(target?, newpri?)\},$
$log\_context := log\_context \oplus$
$\{(running\_task, phys\_context)\},$
$phys\_context := log\_context(topReady!),$

$$
\begin{array}{l}
\quad priority := priority \oplus \{(target?, newpri?)\}, \\
\quad running\_task := topReady!, \\
\quad state := state \oplus \\
\qquad (\{(running\_task, ready)\} \cup \{(topReady!, running)\})] \\
\land\ newpri? < priority(topReady!) \\
\land\ (st \in TASK \\
\quad \land \neg\ (state \oplus (\{(running\_task, ready)\} \cup \\
\qquad \{(topReady!, running)\}))st = state(st) \\
\quad \Rightarrow (state(st), (state \oplus (\{(running\_task, ready)\} \cup \\
\qquad \{(topReady!, running)\}))st) \in transition) \\
\Rightarrow t \in TASK \\
\quad \land\ state(t) = ready \\
\quad \land \neg\ priority(topReady!) \geq priority(t))
\end{array}
$$

theorem ChangeTaskPriorityDNotInherited_TQTM_vc_ref

$\forall\ ChangeTaskPriorityDNotInherited\_TQTMFSBSig \mid true$

• pre $ChangeTaskPriorityDNotInherited\_TQTM$

ChangeTaskPriorityInheritedN_TQTM

$\Delta TaskQueueTimeMutex$
$newpri? : \mathbb{N}$
$target? : TASK$
$topReady! : TASK$

$running\_task \notin \mathrm{dom}\ release\_snd \cup \mathrm{dom}\ release\_rcv$
$target? \in \mathrm{dom}\ base\_priority$
$base\_priority(target?) \neq priority(target?)$
$newpri? \leq priority(target?)$
$state(target?) \neq nonexistent$
$target? = idle \Rightarrow newpri? = 0$
$\Xi TaskQueueTime$
$\Xi MutexData$
$base\_priority' = base\_priority \oplus \{(target? \mapsto newpri?)\}$
$\Xi MReleasingData$
$topReady! = running\_task$

ChangeTaskPriorityInheritedN_TQTMFSBSig

$TaskQueueTimeMutex$
$newpri? : \mathbb{N}$
$target? : TASK$

$running\_task \notin \mathrm{dom}\ release\_snd \cup \mathrm{dom}\ release\_rcv$
$target? \in \mathrm{dom}\ base\_priority$
$base\_priority(target?) \neq priority(target?)$

$newpri? \leq priority(target?)$
$state(target?) \neq nonexistent$
$target? = idle \Rightarrow newpri? = 0$

theorem ChangeTaskPriorityInheritedN_TQTM_vc_ref
$\forall$ *ChangeTaskPriorityInheritedN_TQTMFSBSig* | *true*
• pre *ChangeTaskPriorityInheritedN_TQTM*

ChangeTaskPriorityInheritedU_TQTM
Δ *TaskQueueTimeMutex*
ChangeTaskPriorityN_TQT

$target? \in \text{dom}\ base\_priority$
$base\_priority(target?) \neq priority(target?)$
$newpri? > priority(target?)$
$\Xi MutexData$
$base\_priority' = base\_priority \oplus \{(target? \mapsto newpri?)\}$
$\Xi MReleasingData$

ChangeTaskPriorityInheritedU_TQTMFSBSig
ChangeTaskPriorityN_TQTFSBSig
TaskQueueTimeMutex

$target? \in \text{dom}\ base\_priority$
$base\_priority(target?) \neq priority(target?)$
$newpri? > priority(target?)$

theorem ChangeTaskPriorityInheritedU_TQTM_vc_ref
$\forall$ *ChangeTaskPriorityInheritedU_TQTMFSBSig* | *true*
• pre *ChangeTaskPriorityInheritedU_TQTM*

ChangeTaskPriorityInheritedS_TQTM
Δ *TaskQueueTimeMutex*
ChangeTaskPriorityS_TQT

$target? \in \text{dom}\ base\_priority$
$base\_priority(target?) \neq priority(target?)$
$newpri? > priority(target?)$
$\Xi MutexData$
$base\_priority' = base\_priority \oplus \{(target? \mapsto newpri?)\}$
$\Xi MReleasingData$

ChangeTaskPriorityInheritedS_TQTMFSBSig

$ChangeTaskPriorityS\_TQTFSBSig$
$TaskQueueTimeMutex$

$target? \in \mathrm{dom}\ base\_priority$
$base\_priority(target?) \neq priority(target?)$
$newpri? > priority(target?)$

theorem ChangeTaskPriorityInheritedS_TQTM_vc_ref
$\forall\ ChangeTaskPriorityInheritedS\_TQTMFSBSig \mid true$
• pre $ChangeTaskPriorityInheritedS\_TQTM$

$$
\begin{aligned}
ChangeTaskPriority\_TQTM \mathrel{\widehat=}\ & ChangeTaskPriorityNNotHolder\_TQTM \\
\lor\ & ChangeTaskPrioritySNotHolder\_TQTM \\
\lor\ & ChangeTaskPriorityDNotHolder\_TQTM \\
\lor\ & ChangeTaskPriorityNNotInherited\_TQTM \\
\lor\ & ChangeTaskPrioritySNotInherited\_TQTM \\
\lor\ & ChangeTaskPriorityDNotInherited\_TQTM \\
\lor\ & ChangeTaskPriorityInheritedN\_TQTM \\
\lor\ & ChangeTaskPriorityInheritedU\_TQTM \\
\lor\ & ChangeTaskPriorityInheritedS\_TQTM
\end{aligned}
$$

$$CreateQueue\_TQTM \mathrel{\widehat=} ExtendTQTXi \land CreateQueue\_TQT$$

$$
\begin{aligned}
CreateQueue\_TQTMFSBSig \mathrel{\widehat=}\ & TaskQueueTimeMutex \\
\land\ & CreateQueue\_TQTFSBSig
\end{aligned}
$$

theorem CreateQueue_TQTM_vc_ref
$\forall\ CreateQueue\_TQTMFSBSig \mid true \bullet$ pre $CreateQueue\_TQTM$

DeleteQueue_TQTM

$DeleteQueue\_TQT$
$\Delta TaskQueueTimeMutex$

$que? \notin semaphore \cup mutex$
$\Xi Mutex$

DeleteQueue_TQTMFSBSig

$TaskQueueTimeMutex$
$DeleteQueue\_TQTFSBSig$

$que? \notin semaphore \cup mutex$

theorem DeleteQueue_TQTM_vc_ref
$\forall\, DeleteQueue\_TQTMFSBSig \mid true \bullet \text{pre}\ DeleteQueue\_TQTM$

QueueSendN_TQTM

$QueueSendN\_TQT$
$\Delta TaskQueueTimeMutex$

$que? \notin mutex \cup semaphore$
$\Xi Mutex$

QueueSendN_TQTMFSBSig

$TaskQueueTimeMutex$
$QueueSendN\_TQTFSBSig$

$que? \notin mutex \cup semaphore$

theorem QueueSendN_TQTM_vc_ref
$\forall\, QueueSendN\_TQTMFSBSig \mid true \bullet \text{pre}\ QueueSendN\_TQTM$

QueueSendF_TQTM

$\Delta TaskQueueTimeMutex$
$QueueSendF\_TQT$

$que? \notin mutex \cup semaphore$
$\Xi Mutex$

QueueSendF_TQTMFSBSig

$TaskQueueTimeMutex$
$QueueSendF\_TQTFSBSig$

$que? \notin mutex \cup semaphore$

theorem lQueueSendF_TQTM_Lemma
$\forall\, TaskQueueTimeMutex;\ topReady! : TASK;\ que? : QUEUE;\ wtime? : \mathbb{N}$
$\mid running\_task \notin \text{dom}\ release\_rcv$
$\land\ (running\_task \in \text{dom}\ release\_snd$
$\Rightarrow que? = release\_snd(running\_task))$
$\land\ que? \in queue$
$\land\ q\_size(que?) = q\_max(que?)$
$\land\ running\_task \neq idle$

$$
\begin{array}{l}
\quad \wedge\ state(topReady!) = ready \\
\quad \wedge\ (\forall\, rtsk : state^{\sim}(\!|\ \{ready\}\ |\!) \\
\qquad \bullet\ priority(topReady!) \geq priority(rtsk)) \\
\quad \wedge\ wtime? > clock \\
\quad \wedge\ que? \notin mutex \cup semaphore \\
\quad \bullet\ \neg\ (TaskQueueTimeMutex[log\_context := log\_context \oplus \\
\qquad\qquad \{(running\_task, phys\_context)\}, \\
\qquad\quad phys\_context := log\_context(topReady!), \\
\qquad\quad release\_snd := \{running\_task\} \lhd release\_snd, \\
\qquad\quad running\_task := topReady!, \\
\qquad\quad state := state \oplus \\
\qquad\qquad (\{(running\_task, blocked)\} \cup \{(topReady!, running)\}), \\
\qquad\quad wait\_time := wait\_time \oplus \{(running\_task, wtime?)\}, \\
\qquad\quad wait\_snd := wait\_snd \oplus \{(running\_task, que?)\}] \\
\qquad \wedge\ (st \in TASK \\
\qquad\quad \wedge\ \neg\ (state \oplus (\{(running\_task, blocked)\} \cup \\
\qquad\qquad\quad \{(topReady!, running)\}))st = state(st) \\
\qquad\qquad \Rightarrow (state(st), (state \oplus (\{(running\_task, blocked)\} \cup \\
\qquad\qquad\quad \{(topReady!, running)\}))st) \in transition) \\
\qquad \Rightarrow\ t \in TASK \\
\qquad\quad \wedge\ state(t) = ready \\
\qquad\quad \wedge\ \neg\ priority(topReady!) \geq priority(t))
\end{array}
$$

theorem QueueSendF_TQTM_vc_ref

$\forall\, QueueSendF\_TQTMFSBSig \mid true \bullet \text{pre}\ QueueSendF\_TQTM$

QueueSendW_TQTM

$\Delta\, TaskQueueTimeMutex$

$QueueSendW\_TQT$

$que? \notin mutex \cup semaphore$

$\Xi Mutex$

QueueSendW_TQTMFSBSig

$TaskQueueTimeMutex$

$QueueSendW\_TQTFSBSig$

$que? \notin mutex \cup semaphore$

theorem overrideIsDisjointUnion $[X, Y]$

$\forall\, f, g : X \pfun Y \mid \mathrm{dom}\, f \cap \mathrm{dom}\, g = \emptyset \bullet f \subseteq f \oplus g$

theorem lQueueSendW_TQTM_Lemma

$$
\begin{array}{l}
\forall\, TaskQueueTimeMutex;\ topReady! : TASK;\ que? : QUEUE \\
\quad \mid running\_task \notin \operatorname{dom} release\_rcv \\
\quad \land\ (running\_task \in \operatorname{dom} release\_snd \\
\qquad \Rightarrow que? = release\_snd(running\_task)) \\
\quad \land\ que? \in queue \\
\quad \land\ q\_size(que?) < q\_max(que?) \\
\quad \land\ topReady! \in wait\_rcv^{\sim}(\!|\ \{que?\}\ |\!) \\
\quad \land\ (\forall\, wrct : wait\_rcv^{\sim}(\!|\ \{que?\}\ |\!) \\
\qquad \bullet\ priority(topReady!) \geq priority(wrct)) \\
\quad \land\ priority(running\_task) \geq priority(topReady!) \\
\quad \land\ que? \notin mutex \cup semaphore \\
\quad \bullet\ \neg\ (TaskQueueTimeMutex[q\_size := q\_size \oplus \{(que?, (1+ \\
\qquad\qquad q\_size(que?)))\}, \\
\qquad\qquad release\_rcv := release\_rcv \oplus \{(topReady!, que?)\}, \\
\qquad\qquad release\_snd := \{running\_task\} \lhd release\_snd, \\
\qquad\qquad state := state \oplus \{(topReady!, ready)\}, \\
\qquad\qquad wait\_time := \{topReady!\} \lhd wait\_time, \\
\qquad\qquad wait\_rcv := \{topReady!\} \lhd wait\_rcv] \\
\qquad \land\ priority(topReady!) \leq priority(running\_task) \\
\qquad \land\ (st \in TASK \land \neg\ (state \oplus \{(topReady!, ready)\})st = state(st) \\
\qquad\qquad \Rightarrow (state(st), (state \oplus \{(topReady!, ready)\})st) \\
\qquad\qquad\qquad \in transition) \\
\qquad \Rightarrow wr \in \operatorname{dom} wait\_rcv \\
\qquad\qquad \land\ wait\_rcv(wr) = que? \\
\qquad\qquad \land\ \neg\ priority(topReady!) \geq priority(wr))
\end{array}
$$

theorem QueueSendW_TQTM_vc_ref

$$\forall\, QueueSendW\_TQTMFSBSig \mid true \bullet \text{pre}\ QueueSendW\_TQTM$$

QueueSendWS_TQTM

$$
\begin{array}{l}
\Delta\, TaskQueueTimeMutex \\
QueueSendWS\_TQT \\
\hline
que? \notin mutex \cup semaphore \\
\Xi Mutex
\end{array}
$$

QueueSendWS_TQTMFSBSig

$$
\begin{array}{l}
TaskQueueTimeMutex \\
QueueSendWS\_TQTFSBSig \\
\hline
que? \notin mutex \cup semaphore
\end{array}
$$

theorem lQueueSendWS_TQTM_Lemma

$$
\begin{array}{l}
\forall\ TaskQueueTimeMutex;\ topReady! : TASK;\ que? : QUEUE \\
\quad \mid running\_task \notin \operatorname{dom} release\_rcv \\
\quad \land\ (running\_task \in \operatorname{dom} release\_snd \\
\qquad \Rightarrow que? = release\_snd(running\_task)) \\
\quad \land\ que? \in queue \\
\quad \land\ q\_size(que?) < q\_max(que?) \\
\quad \land\ topReady! \in wait\_rcv^{\sim}(\!|\ \{que?\}\ |\!) \\
\quad \land\ (\forall\ wrct : wait\_rcv^{\sim}(\!|\ \{que?\}\ |\!) \\
\qquad \bullet\ priority(topReady!) \geq priority(wrct)) \\
\quad \land\ priority(topReady!) > priority(running\_task) \\
\quad \land\ que? \notin mutex \cup semaphore \\
\quad \bullet \neg\ (TaskQueueTimeMutex[log\_context := log\_context \oplus \\
\qquad\qquad \{(running\_task, phys\_context)\}, \\
\qquad\quad phys\_context := log\_context(topReady!), \\
\qquad\quad q\_size := q\_size \oplus \{(wait\_rcv(topReady!), (1+ \\
\qquad\qquad q\_size(wait\_rcv(topReady!))))\}, \\
\qquad\quad release\_rcv := release\_rcv \oplus \\
\qquad\qquad \{(topReady!, wait\_rcv(topReady!))\}, \\
\qquad\quad release\_snd := \{running\_task\} \lhd release\_snd, \\
\qquad\quad running\_task := topReady!, \\
\qquad\quad state := state \oplus \\
\qquad\qquad (\{(running\_task, ready)\} \cup \{(topReady!, running)\}), \\
\qquad\quad wait\_time := \{topReady!\} \lhd wait\_time, \\
\qquad\quad wait\_rcv := \{topReady!\} \lhd wait\_rcv] \\
\qquad \land\ (st \in TASK \\
\qquad\quad \land \neg\ (state \oplus (\{(running\_task, ready)\} \cup \\
\qquad\qquad\quad \{(topReady!, running)\}))st = state(st) \\
\qquad\quad \Rightarrow (state(st), (state \oplus (\{(running\_task, ready)\} \cup \\
\qquad\qquad\quad \{(topReady!, running)\}))st) \in transition) \\
\qquad \Rightarrow wr \in \operatorname{dom} wait\_rcv \\
\qquad\quad \land\ wait\_rcv(wr) = wait\_rcv(topReady!) \\
\qquad\quad \land \neg\ priority(topReady!) \geq priority(wr))
\end{array}
$$

theorem QueueSendWS_TQTM_vc_ref

$$\forall\ QueueSendWS\_TQTMFSBSig \mid true \bullet \text{pre}\ QueueSendWS\_TQTM$$

$$
\begin{array}{l}
QueueSend\_TQTM \mathrel{\widehat=} QueueSendN\_TQTM \\
\qquad\qquad\qquad \lor\ QueueSendF\_TQTM \\
\qquad\qquad\qquad \lor\ QueueSendW\_TQTM \\
\qquad\qquad\qquad \lor\ QueueSendWS\_TQTM
\end{array}
$$

QueueReceiveN_TQTM

$$
\begin{array}{l}
QueueReceiveN\_TQT \\
\Delta TaskQueueTimeMutex
\end{array}
$$

$$
\begin{array}{|l}
\hline
running\_task \notin \mathrm{dom}\ release\_mutex \\
que? \notin mutex \cup semaphore \\
\Xi Mutex \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
QueueReceiveN\_TQTMFSBSig \\
\hline
TaskQueueTimeMutex \\
QueueReceiveN\_TQTFSBSig \\
\hline
running\_task \notin \mathrm{dom}\ release\_mutex \\
que? \notin mutex \cup semaphore \\
\hline
\end{array}
$$

theorem mutexDiffQue

$$
\forall\ TaskQueueTimeMutex \mid que? \notin mutex \cup semaphore \bullet \mathrm{dom}\ mutex\_holder = \{f : mutex \mid (q\_size \oplus \{(que?, (q\_size(que?) - 1))\})f = 0\}
$$

theorem QueueReceiveN_TQTM_vc_ref

$$
\forall\ QueueReceiveN\_TQTMFSBSig \mid true \bullet \mathrm{pre}\ QueueReceiveN\_TQTM
$$

$$
\begin{array}{|l}
QueueReceiveE\_TQTM \\
\hline
\Delta TaskQueueTimeMutex \\
QueueReceiveE\_TQT \\
\hline
running\_task \notin \mathrm{dom}\ release\_mutex \\
que? \notin mutex \cup semaphore \\
\Xi Mutex \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
QueueReceiveE\_TQTMFSBSig \\
\hline
TaskQueueTimeMutex \\
QueueReceiveE\_TQTFSBSig \\
\hline
running\_task \notin \mathrm{dom}\ release\_mutex \\
que? \notin mutex \cup semaphore \\
\hline
\end{array}
$$

theorem lQueueReceiveE_TQTM_Lemma

$$
\begin{array}{l}
\forall\ TaskQueueTimeMutex;\ que? : QUEUE;\ topReady! : TASK;\ wtime? : \mathbb{N} \\
\quad \mid running\_task \notin \mathrm{dom}\ release\_snd \\
\quad \land\ (running\_task \in \mathrm{dom}\ release\_rcv \\
\qquad \Rightarrow que? = release\_rcv(running\_task)) \\
\quad \land\ que? \in queue \\
\quad \land\ q\_size(que?) = 0
\end{array}
$$

$$
\begin{array}{l}
\quad \wedge\ running\_task \neq idle \\
\quad \wedge\ topReady! \in state^{\sim}(\!|\ \{ready\}\ |\!) \\
\quad \wedge\ (\forall\, rtsk : state^{\sim}(\!|\ \{ready\}\ |\!) \\
\qquad \bullet\ priority(topReady!) \geq priority(rtsk)) \\
\quad \wedge\ wtime? > clock \\
\quad \wedge\ running\_task \notin \mathrm{dom}\, release\_mutex \\
\quad \wedge\ que? \notin mutex \cup semaphore \\
\quad \bullet \neg\ (TaskQueueTimeMutex[log\_context := log\_context \oplus \\
\qquad\qquad \{(running\_task, phys\_context)\}, \\
\qquad\quad phys\_context := log\_context(topReady!), \\
\qquad\quad release\_rcv := \{running\_task\} \lhd release\_rcv, \\
\qquad\quad running\_task := topReady!, \\
\qquad\quad state := state \oplus \\
\qquad\qquad (\{(running\_task, blocked)\} \cup \{(topReady!, running)\}), \\
\qquad\quad wait\_time := wait\_time \oplus \{(running\_task, wtime?)\}, \\
\qquad\quad wait\_rcv := wait\_rcv \oplus \{(running\_task, que?)\}] \\
\qquad \wedge\ (st \in TASK \\
\qquad\quad \wedge \neg\ (state \oplus (\{(running\_task, blocked)\} \cup \\
\qquad\qquad\quad \{(topReady!, running)\}))st = state(st) \\
\qquad\quad\ \Rightarrow (state(st), (state \oplus (\{(running\_task, blocked)\} \cup \\
\qquad\qquad\quad \{(topReady!, running)\}))st) \in transition) \\
\qquad \Rightarrow t \in TASK \\
\qquad\quad \wedge\ state(t) = ready \\
\qquad\quad \wedge \neg\ priority(topReady!) \geq priority(t))
\end{array}
$$

theorem QueueReceiveE_TQTM_vc_ref

$\forall\, QueueReceiveE\_TQTMFSBSig \mid true \bullet \mathrm{pre}\ QueueReceiveE\_TQTM$

QueueReceiveW_TQTM

$\Delta TaskQueueTimeMutex$

$QueueReceiveW\_TQT$

$running\_task \notin \mathrm{dom}\, release\_mutex$

$que? \notin mutex \cup semaphore$

$\Xi Mutex$

QueueReceiveW_TQTMFSBSig

$TaskQueueTimeMutex$

$QueueReceiveW\_TQTFSBSig$

$running\_task \notin \mathrm{dom}\, release\_mutex$

$que? \notin mutex \cup semaphore$

theorem lQueueReceiveW_TQTM_Lemma

$$
\begin{array}{l}
\forall\ TaskQueueTimeMutex;\ que? : QUEUE;\ topReady! : TASK \\
\quad \mid running\_task \notin \operatorname{dom} release\_snd \\
\quad \land\ (running\_task \in \operatorname{dom} release\_rcv \\
\qquad \Rightarrow que? = release\_rcv(running\_task)) \\
\quad \land\ que? \in queue \\
\quad \land\ q\_size(que?) > 0 \\
\quad \land\ topReady! \in wait\_snd^{\sim}(\!|\ \{que?\}\ |\!) \\
\quad \land\ (\forall\ wsnt : wait\_snd^{\sim}(\!|\ \{que?\}\ |\!) \\
\qquad \bullet\ priority(topReady!) \geq priority(wsnt)) \\
\quad \land\ priority(running\_task) \geq priority(topReady!) \\
\quad \land\ running\_task \notin \operatorname{dom} release\_mutex \\
\quad \land\ que? \notin mutex \cup semaphore \\
\quad \bullet \neg\ (TaskQueueTimeMutex[q\_size := q\_size \oplus \\
\qquad\qquad \{(que?, (q\_size(que?) - 1))\}, \\
\qquad\quad release\_rcv := \{running\_task\} \lhd release\_rcv, \\
\qquad\quad release\_snd := release\_snd \oplus \{(topReady!, que?)\}, \\
\qquad\quad state := state \oplus \{(topReady!, ready)\}, \\
\qquad\quad wait\_time := \{topReady!\} \lhd wait\_time, \\
\qquad\quad wait\_snd := \{topReady!\} \lhd wait\_snd] \\
\qquad \land\ priority(topReady!) \leq priority(running\_task) \\
\qquad \land\ (st \in TASK \\
\qquad\quad \land \neg\ (state \oplus \{(topReady!, ready)\})st = state(st) \\
\qquad\qquad \Rightarrow (state(st), (state \oplus \{(topReady!, ready)\})st) \\
\qquad\qquad\quad \in transition) \\
\qquad \Rightarrow ws \in \operatorname{dom} wait\_snd \\
\qquad\quad \land\ wait\_snd(ws) = que? \\
\qquad\quad \land \neg\ priority(topReady!) \geq priority(ws))
\end{array}
$$

theorem QueueReceiveW_TQTM_vc_ref

$$\forall\ QueueReceiveW\_TQTMFSBSig \mid true \bullet \text{pre}\ QueueReceiveW\_TQTM$$

QueueReceiveWS_TQTM

$$
\begin{array}{l}
\Delta TaskQueueTimeMutex \\
QueueReceiveWS\_TQT \\
\hline
running\_task \notin \operatorname{dom} release\_mutex \\
que? \notin mutex \cup semaphore \\
\Xi Mutex
\end{array}
$$

QueueReceiveWS_TQTMFSBSig

$$
\begin{array}{l}
TaskQueueTimeMutex \\
QueueReceiveWS\_TQTFSBSig \\
\hline
\end{array}
$$

$running\_task \notin \mathrm{dom}\, release\_mutex$
$que? \notin mutex \cup semaphore$

---

theorem lQueueReceiveWS_TQTM_Lemma

$\forall\ TaskQueueTimeMutex;\ que? : QUEUE;\ topReady! : TASK$
$\quad \mid running\_task \notin \mathrm{dom}\, release\_snd$
$\quad \wedge\ (running\_task \in \mathrm{dom}\, release\_rcv$
$\quad\quad \Rightarrow que? = release\_rcv(running\_task))$
$\quad \wedge\ que? \in queue$
$\quad \wedge\ q\_size(que?) > 0$
$\quad \wedge\ topReady! \in wait\_snd^{\sim}(\!|\ \{que?\}\ |\!)$
$\quad \wedge\ (\forall\ wsnt : wait\_snd^{\sim}(\!|\ \{que?\}\ |\!)$
$\quad\quad \bullet\ priority(topReady!) \geq priority(wsnt))$
$\quad \wedge\ priority(topReady!) > priority(running\_task)$
$\quad \wedge\ running\_task \notin \mathrm{dom}\, release\_mutex$
$\quad \wedge\ que? \notin mutex \cup semaphore$
$\quad \bullet\ \neg\ (TaskQueueTimeMutex[log\_context := log\_context \oplus$
$\quad\quad\quad \{(running\_task, phys\_context)\},$
$\quad\quad phys\_context := log\_context(topReady!),$
$\quad\quad q\_size := q\_size \oplus \{(wait\_snd(topReady!),$
$\quad\quad\quad (q\_size(wait\_snd(topReady!)) - 1))\},$
$\quad\quad release\_rcv := \{running\_task\} \lhd release\_rcv,$
$\quad\quad release\_snd := release\_snd \oplus$
$\quad\quad\quad \{(topReady!, wait\_snd(topReady!))\},$
$\quad\quad running\_task := topReady!,$
$\quad\quad state := state \oplus$
$\quad\quad\quad (\{(running\_task, ready)\} \cup \{(topReady!, running)\}),$
$\quad\quad wait\_time := \{topReady!\} \lhd wait\_time,$
$\quad\quad wait\_snd := \{topReady!\} \lhd wait\_snd]$
$\quad\quad \wedge\ (st \in TASK$
$\quad\quad\quad \wedge\ \neg\ (state \oplus (\{(running\_task, ready)\} \cup$
$\quad\quad\quad\quad \{(topReady!, running)\}))st = state(st)$
$\quad\quad\quad\quad \Rightarrow (state(st), (state \oplus (\{(running\_task, ready)\} \cup$
$\quad\quad\quad\quad\quad \{(topReady!, running)\}))st) \in transition)$
$\quad\quad \Rightarrow ws \in \mathrm{dom}\, wait\_snd$
$\quad\quad\quad \wedge\ wait\_snd(ws) = wait\_snd(topReady!)$
$\quad\quad\quad \wedge\ \neg\ priority(topReady!) \geq priority(ws))$

theorem QueueReceiveWS_TQTM_vc_ref

$\forall\ QueueReceiveWS\_TQTMFSBSig \mid true \bullet \mathrm{pre}\ QueueReceiveWS\_TQTM$

$QueueReceive\_TQTM \mathrel{\widehat=} QueueReceiveN\_TQTM$
$\quad\quad\quad\quad\quad\quad \vee\ QueueReceiveE\_TQTM$
$\quad\quad\quad\quad\quad\quad \vee\ QueueReceiveW\_TQTM$

$$\vee\ QueueReceiveWS\_TQTM$$

$$DelayUntil\_TQTM \mathrel{\widehat=} ExtendTQTXi \wedge DelayUntil\_TQT$$

$$DelayUntil\_TQTMFSBSig \mathrel{\widehat=} TaskQueueTimeMutex \wedge DelayUntil\_TQTFSBSig$$

theorem lDelayUntil_TQTM_Lemma

$$\begin{array}{l}
\forall\ TaskQueueTimeMutex;\ wtime? : \mathbb{N};\ topReady! : TASK \\
\quad \mid running_task \notin \operatorname{dom} release_snd \cup \operatorname{dom} release_rcv \\
\quad \wedge\ running_task \neq idle \\
\quad \wedge\ state(topReady!) = ready \\
\quad \wedge\ wtime? > clock \\
\quad \wedge\ (\forall\ rtsk : state^{\sim}(\!|\ \{ready\}\ |\!) \\
\qquad \bullet\ priority(topReady!) \geq priority(rtsk)) \\
\quad \bullet\ \neg\ (TaskQueueTimeMutex[delayed_task := delayed_task \cup \\
\qquad\qquad \{running_task\}, \\
\qquad log_context := log_context \oplus \\
\qquad\qquad \{(running_task, phys_context)\}, \\
\qquad phys_context := log_context(topReady!), \\
\qquad running_task := topReady!, \\
\qquad state := state \oplus \\
\qquad\qquad (\{(running_task, blocked)\} \cup \{(topReady!, running)\}), \\
\qquad wait_time := wait_time \oplus \{(running_task, wtime?)\}] \\
\qquad \wedge\ (st \in TASK \\
\qquad\quad \wedge\ \neg\ (state \oplus (\{(running_task, blocked)\} \cup \\
\qquad\qquad\qquad \{(topReady!, running)\}))st = state(st) \\
\qquad\quad \Rightarrow (state(st), (state \oplus (\{(running_task, blocked)\} \cup \\
\qquad\qquad\qquad \{(topReady!, running)\}))st) \in transition) \\
\qquad \Rightarrow t \in TASK \\
\qquad\quad \wedge\ state(t) = ready \\
\qquad\quad \wedge\ \neg\ priority(topReady!) \geq priority(t))
\end{array}$$

theorem DelayUntil_TQTM_vc_ref

$$\forall\ DelayUntil\_TQTMFSBSig \mid true \bullet \text{pre}\ DelayUntil\_TQTM$$

$$CheckDelayedTaskN\_TQTM \mathrel{\widehat=} ExtendTQTXi \wedge CheckDelayedTaskN\_TQT$$

$$CheckDelayedTaskN\_TQTMFSBSig \mathrel{\widehat=} TaskQueueTimeMutex \wedge CheckDelayedTaskN\_TQTFSBSig$$

theorem lCheckDelayedTaskN_TQTM_Lemma

$$
\begin{array}{l}
\forall\ TaskQueueTimeMutex;\ topWaiting! : TASK \\
\quad \mid running\_task \notin \mathrm{dom}\ release\_snd \cup \mathrm{dom}\ release\_rcv \\
\quad \land\ topWaiting! \in \mathrm{dom}\ wait\_time \\
\quad \land\ (\forall\ dtk : \mathrm{dom}\ wait\_time \\
\qquad \bullet\ wait\_time(topWaiting!) \leq wait\_time(dtk)) \\
\quad \land\ (\forall\ detk : \mathrm{dom}\ wait\_time \\
\qquad \mid wait\_time(detk) = wait\_time(topWaiting!) \\
\qquad\quad \bullet\ priority(topWaiting!) \geq priority(detk)) \\
\quad \land\ priority(topWaiting!) \leq priority(running\_task) \\
\quad \bullet \neg\ (TaskQueueTimeMutex[clock := wait\_time(topWaiting!), \\
\qquad\quad delayed\_task := delayed\_task \setminus \{topWaiting!\}, \\
\qquad\quad state := state \oplus \{(topWaiting!, ready)\}, \\
\qquad\quad wait\_time := \{topWaiting!\} \lhd wait\_time, \\
\qquad\quad wait\_rcv := \{topWaiting!\} \lhd wait\_rcv, \\
\qquad\quad wait\_snd := \{topWaiting!\} \lhd wait\_snd] \\
\qquad \land\ (st \in TASK \\
\qquad\quad \land \neg\ (state \oplus \{(topWaiting!, ready)\})st = state(st) \\
\qquad\quad\quad \Rightarrow (state(st), (state \oplus \{(topWaiting!, ready)\})st) \\
\qquad\quad\quad\quad \in transition) \\
\qquad \land\ (wt \in \mathrm{dom}\ wait\_time \\
\qquad\quad \Rightarrow wait\_time(topWaiting!) \leq wait\_time(wt)) \\
\qquad \Rightarrow wt\_\_0 \in \mathrm{dom}\ wait\_time \\
\qquad\quad \land\ wait\_time(wt\_\_0) = wait\_time(topWaiting!) \\
\qquad\quad \land \neg\ priority(topWaiting!) \geq priority(wt\_\_0))
\end{array}
$$

theorem CheckDelayedTaskN_TQTM_vc_ref

$$
\begin{array}{l}
\forall\ CheckDelayedTaskN\_TQTMFSBSig \mid true \\
\quad \bullet\ \mathrm{pre}\ CheckDelayedTaskN\_TQTM
\end{array}
$$

$$CheckDelayedTaskS\_TQTM \mathrel{\widehat=} ExtendTQTXi \land CheckDelayedTaskS\_TQT$$

$$
\begin{array}{l}
CheckDelayedTaskS\_TQTMFSBSig \mathrel{\widehat=} TaskQueueTimeMutex \\
\qquad\qquad\qquad\qquad\qquad\qquad \land\ CheckDelayedTaskS\_TQTFSBSig
\end{array}
$$

theorem lCheckDelayedTaskS_TQTM_Lemma

$$
\begin{array}{l}
\forall\ TaskQueueTimeMutex;\ topWaiting! : TASK \\
\quad \mid running\_task \notin \mathrm{dom}\ release\_snd \cup \mathrm{dom}\ release\_rcv \\
\quad \land\ topWaiting! \in \mathrm{dom}\ wait\_time \\
\quad \land\ (\forall\ dtk : \mathrm{dom}\ wait\_time \\
\qquad \bullet\ wait\_time(topWaiting!) \leq wait\_time(dtk)) \\
\quad \land\ (\forall\ detk : \mathrm{dom}\ wait\_time \\
\qquad \mid wait\_time(detk) = wait\_time(topWaiting!) \\
\qquad\quad \bullet\ priority(topWaiting!) \geq priority(detk))
\end{array}
$$

$$
\begin{array}{l}
\quad \wedge\ priority(topWaiting!) > priority(running\_task) \\
\quad \bullet\ \neg\ (wait\_time(topWaiting!) \in \mathbb{Z} \\
\qquad \wedge\ TaskQueueTimeMutex[clock := wait\_time(topWaiting!), \\
\qquad\quad delayed\_task := delayed\_task \setminus \{topWaiting!\}, \\
\qquad\quad log\_context := log\_context\oplus \\
\qquad\qquad \{(running\_task, phys\_context)\}, \\
\qquad\quad phys\_context := log\_context(topWaiting!), \\
\qquad\quad running\_task := topWaiting!, \\
\qquad\quad state := state\oplus \\
\qquad\qquad (\{(running\_task, ready)\} \cup \{(topWaiting!, running)\}), \\
\qquad\quad wait\_time := \{topWaiting!\} \lhd wait\_time, \\
\qquad\quad wait\_rcv := \{topWaiting!\} \lhd wait\_rcv, \\
\qquad\quad wait\_snd := \{topWaiting!\} \lhd wait\_snd] \\
\qquad \wedge\ (st \in TASK \\
\qquad\quad \wedge\ \neg\ (state \oplus (\{(running\_task, ready)\} \cup \\
\qquad\qquad\qquad \{(topWaiting!, running)\}))st = state(st) \\
\qquad\qquad \Rightarrow (state(st), (state \oplus (\{(running\_task, ready)\} \cup \\
\qquad\qquad\qquad \{(topWaiting!, running)\}))st) \in transition) \\
\qquad \wedge\ (wt \in \mathrm{dom}\ wait\_time \\
\qquad\quad \Rightarrow wait\_time(topWaiting!) \leq wait\_time(wt)) \\
\qquad \Rightarrow wt\_\_0 \in \mathrm{dom}\ wait\_time \\
\qquad\quad \wedge\ wait\_time(wt\_\_0) = wait\_time(topWaiting!) \\
\qquad\quad \wedge\ \neg\ priority(topWaiting!) \geq priority(wt\_\_0))
\end{array}
$$

theorem CheckDelayedTaskS_TQTM_vc_ref

$$
\begin{array}{l}
\quad \forall\ CheckDelayedTaskS\_TQTMFSBSig \mid true \\
\qquad \bullet\ \mathrm{pre}\ CheckDelayedTaskS\_TQTM
\end{array}
$$

$$
TimeSlicing\_TQTM \mathrel{\widehat=} ExtendTQTXi \wedge TimeSlicing\_TQT
$$

$$
\begin{array}{l}
TimeSlicing\_TQTMFSBSig \mathrel{\widehat=} TaskQueueTimeMutex \\
\qquad\qquad\qquad\qquad\qquad \wedge\ TimeSlicing\_TQTFSBSig
\end{array}
$$

theorem lTimeSlicing_TQTM_Lemma

$$
\begin{array}{l}
\quad \forall\ TaskQueueTimeMutex;\ topReady! : TASK \\
\qquad \mid running\_task \notin \mathrm{dom}\ release\_snd \cup \mathrm{dom}\ release\_rcv \\
\qquad \wedge\ state(topReady!) = ready \\
\qquad \wedge\ priority(topReady!) = priority(running\_task) \\
\qquad \wedge\ (\forall\ ts : \mathrm{dom}\ wait\_time \bullet time\_slice \leq wait\_time(ts)) \\
\qquad \bullet\ \neg\ (TaskQueueTimeMutex[log\_context := log\_context\oplus \\
\qquad\qquad\qquad \{(running\_task, phys\_context)\}, \\
\qquad\qquad phys\_context := log\_context(topReady!), \\
\qquad\qquad running\_task := topReady!, \\
\qquad\qquad state := state\oplus
\end{array}
$$

$$
\begin{array}{l}
\qquad (\{(running\_task, ready)\} \cup \{(topReady!, running)\}), \\
\quad time\_slice := time\_slice + slice\_delay] \\
\Rightarrow st \in TASK \\
\quad \land \neg\, (state \oplus (\{(running\_task, ready)\} \cup \\
\qquad \{(topReady!, running)\}))st = state(st) \\
\quad \land \neg\, (state(st), (state \oplus (\{(running\_task, ready)\} \cup \\
\qquad \{(topReady!, running)\}))st) \in transition)
\end{array}
$$

theorem TimeSlicing_TQTM_vc_ref

$\forall\, TimeSlicing\_TQTMFSBSig \mid true \bullet \text{pre}\ TimeSlicing\_TQTM$

$$NoSlicing\_TQTM \mathrel{\widehat=} ExtendTQTXi \land NoSlicing\_TQT$$

$$
\begin{array}{l}
NoSlicing\_TQTMFSBSig \mathrel{\widehat=} TaskQueueTimeMutex \\
\qquad\qquad\qquad\qquad \land\ NoSlicing\_TQTFSBSig
\end{array}
$$

theorem NoSlicing_TQTM_vc_ref

$\forall\, NoSlicing\_TQTMFSBSig \mid true \bullet \text{pre}\ NoSlicing\_TQTM$

CreateBinarySemaphore_TQTM

$\Delta TaskQueueTimeMutex$
$sem? : QUEUE$

$running\_task \notin \text{dom}\ release\_snd \cup \text{dom}\ release\_rcv$
$sem? \notin queue$
$\Xi Task$
$queue' = queue \cup \{sem?\}$
$q\_max' = q\_max \oplus \{(sem? \mapsto 1)\}$
$q\_size' = q\_size \oplus \{(sem? \mapsto 1)\}$
$\Xi WaitingData$
$\Xi QReleasingData$
$\Xi Time$
$semaphore' = semaphore \cup \{sem?\}$
$mutex' = mutex$
$mutex\_holder' = mutex\_holder$
$mutex\_recursive' = mutex\_recursive$
$\Xi OriginalPrioData$
$\Xi MReleasingData$

CreateBinarySemaphore_TQTMFSBSig

$TaskQueueTimeMutex$
$sem? : QUEUE$

$$
\begin{array}{|l}
\hline
running\_task \notin \mathrm{dom}\, release\_snd \cup \mathrm{dom}\, release\_rcv \\
sem? \notin queue \\
\hline
\end{array}
$$

theorem CreateBinarySemaphore_TQTM_vc_ref

$\forall\ CreateBinarySemaphore\_TQTMFSBSig \mid true$

• pre $CreateBinarySemaphore\_TQTM$

$$
\begin{array}{|l}
\hline
DeleteBinarySemaphore\_TQTM \\
\Delta TaskQueueTimeMutex \\
sem? : QUEUE \\
\hline
sem? \in semaphore \\
DeleteQueue\_TQT[sem?/que?] \\
semaphore' = semaphore \setminus \{sem?\} \\
mutex' = mutex \\
mutex\_holder' = mutex\_holder \\
mutex\_recursive' = mutex\_recursive \\
\Xi OriginalPrioData \\
\Xi MReleasingData \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
DeleteBinarySemaphore\_TQTMFSBSig \\
DeleteQueue\_TQTFSBSig[sem?/que?] \\
TaskQueueTimeMutex \\
sem? : QUEUE \\
\hline
sem? \in semaphore \\
\hline
\end{array}
$$

theorem DeleteBinarySemaphore_TQTM_vc_ref

$\forall\ DeleteBinarySemaphore\_TQTMFSBSig \mid true$

• pre $DeleteBinarySemaphore\_TQTM$

$$
\begin{array}{|l}
\hline
CreateMutex\_TQTM \\
\Delta TaskQueueTimeMutex \\
mut? : QUEUE \\
\hline
running\_task \notin \mathrm{dom}\, release\_snd \cup \mathrm{dom}\, release\_rcv \\
mut? \notin queue \\
\Xi Task \\
queue' = queue \cup \{mut?\} \\
q\_max' = q\_max \oplus \{(mut? \mapsto 1)\} \\
q\_size' = q\_size \oplus \{(mut? \mapsto 1)\} \\
\Xi WaitingData \\
\end{array}
$$

$\Xi QReleasingData$
$\Xi Time$
$semaphore' = semaphore$
$mutex' = mutex \cup \{mut?\}$
$mutex\_holder' = mutex\_holder$
$mutex\_recursive' = mutex\_recursive \oplus \{(mut? \mapsto 0)\}$
$\Xi OriginalPrioData$
$\Xi MReleasingData$

CreateMutex_TQTMFSBSig

$TaskQueueTimeMutex$
$mut? : QUEUE$

$running\_task \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv$
$mut? \notin queue$

theorem CreateMutex_TQTM_vc_ref
$\forall CreateMutex\_TQTMFSBSig \mid true \bullet \operatorname{pre} CreateMutex\_TQTM$

DeleteMutex_TQTM

$\Delta TaskQueueTimeMutex$
$mut? : QUEUE$

$mut? \in mutex \setminus \operatorname{dom} mutex\_holder$
$DeleteQueue\_TQT[que? := mut?]$
$semaphore' = semaphore$
$mutex' = mutex \setminus \{mut?\}$
$mutex\_holder' = mutex\_holder$
$mutex\_recursive' = \{mut?\} \lhd mutex\_recursive$
$\Xi OriginalPrioData$
$\Xi MReleasingData$

DeleteMutex_TQTMFSBSig

$TaskQueueTimeMutex$
$mut? : QUEUE$

$DeleteQueue\_TQTFSBSig[que? := mut?]$
$mut? \in mutex \setminus \operatorname{dom} mutex\_holder$

theorem subPfun $[X, Y]$
$\forall f, g : X \pfun Y; \; y : Y \mid g \subseteq f \bullet y \notin \operatorname{ran} f \Rightarrow y \notin \operatorname{ran} g$

theorem DeleteMutex_TQTM_vc_ref

$\forall$ *DeleteMutex_TQTMFSBSig* | *true* • pre *DeleteMutex_TQTM*

MutexTakeNnonInh_TQTM

Δ *TaskQueueTimeMutex*
mut? : *QUEUE*
topReady! : *TASK*

---

running_task $\in$ dom *release_rcv* $\Rightarrow$ *running_task* $\in$ dom *release_mutex*
mut? $\in$ *mutex*
running_task $\notin$ dom *base_priority*
QueueReceiveN_TQT[*que*? := *mut*?]
semaphore$'$ = *semaphore*
mutex$'$ = *mutex*
mutex_holder$'$ = *mutex_holder* $\oplus$ {(*mut*? $\mapsto$ *running_task*)}
mutex_recursive$'$ = *mutex_recursive*$\oplus$
{(*mut*? $\mapsto$ *mutex_recursive*(*mut*?) + 1)}
base_priority$'$ = *base_priority*$\oplus$
{(*running_task* $\mapsto$ *priority*(*running_task*))}
release_mutex$'$ = {*running_task*} $\lhd$ *release_mutex*

MutexTakeNnonInh_TQTMFSBSig

TaskQueueTimeMutex
mut? : *QUEUE*

QueueReceiveN_TQTFSBSig[*que*? := *mut*?]
running_task $\in$ dom *release_rcv* $\Rightarrow$ *running_task* $\in$ dom *release_mutex*
mut? $\in$ *mutex*
running_task $\notin$ dom *base_priority*

theorem MutexTakeNnonInh_TQTM_vc_ref
$\forall$ *MutexTakeNnonInh_TQTMFSBSig* | *true*
• pre *MutexTakeNnonInh_TQTM*

MutexTakeNInh_TQTM

Δ *TaskQueueTimeMutex*
mut? : *QUEUE*
topReady! : *TASK*

---

running_task $\in$ dom *release_rcv* $\Rightarrow$ *running_task* $\in$ dom *release_mutex*
mut? $\in$ *mutex*
running_task $\in$ dom *base_priority*
QueueReceiveN_TQT[*que*? := *mut*?]
semaphore$'$ = *semaphore*
mutex$'$ = *mutex*

$$
\begin{array}{l}
mutex\_holder' = mutex\_holder \oplus \{(mut? \mapsto running\_task)\} \\
mutex\_recursive' = mutex\_recursive \oplus \\
\quad \{(mut? \mapsto mutex\_recursive(mut?) + 1)\} \\
\Xi OriginalPrioData \\
release\_mutex' = \{running\_task\} \lhd release\_mutex
\end{array}
$$

MutexTakeNInh_TQTMFSBSig

$$
\begin{array}{l}
TaskQueueTimeMutex \\
mut? : QUEUE \\
\hline
QueueReceiveN\_TQTFSBSig[que? := mut?] \\
running\_task \in \mathrm{dom}\, release\_rcv \Rightarrow running\_task \in \mathrm{dom}\, release\_mutex \\
mut? \in mutex \\
running\_task \in \mathrm{dom}\, base\_priority
\end{array}
$$

theorem MutexTakeNInh_TQTM_vc_ref

$$\forall MutexTakeNInh\_TQTMFSBSig \mid true \bullet \mathrm{pre}\ MutexTakeNInh\_TQTM$$

MutexTakeEnonInh_TQTM

$$
\begin{array}{l}
\Delta TaskQueueTimeMutex \\
mut? : QUEUE \\
topReady! : TASK \\
wtime? : \mathbb{N} \\
\hline
running\_task \in \mathrm{dom}\, release\_rcv \Rightarrow running\_task \in \mathrm{dom}\, release\_mutex \\
mut? \in \mathrm{dom}\, mutex\_holder \\
priority(running\_task) \leq priority(mutex\_holder(mut?)) \\
running\_task \neq mutex\_holder(mut?) \\
QueueReceiveE\_TQT[que? := mut?] \\
\Xi MutexData \\
\Xi OriginalPrioData \\
release\_mutex' = \{running\_task\} \lhd release\_mutex
\end{array}
$$

MutexTakeEnonInh_TQTMFSBSig

$$
\begin{array}{l}
TaskQueueTimeMutex \\
mut? : QUEUE \\
wtime? : \mathbb{N} \\
\hline
QueueReceiveE\_TQTFSBSig[que? := mut?] \\
running\_task \in \mathrm{dom}\, release\_rcv \Rightarrow running\_task \in \mathrm{dom}\, release\_mutex \\
mut? \in \mathrm{dom}\, mutex\_holder \\
priority(running\_task) \leq priority(mutex\_holder(mut?)) \\
running\_task \neq mutex\_holder(mut?)
\end{array}
$$

theorem lMutexTakeEnonInh_TQTM_Lemma

$$
\begin{array}{l}
\forall\, TaskQueueTimeMutex;\ mut? : QUEUE;\ topReady! : TASK; \\
\quad wtime? : \mathbb{N} \\
\quad \mid running\_task \notin \operatorname{dom} release\_snd \\
\quad \land\ (running\_task \in \operatorname{dom} release\_rcv \\
\qquad \Rightarrow mut? = release\_rcv(running\_task)) \\
\quad \land\ mut? \in queue \\
\quad \land\ q\_size(mut?) = 0 \\
\quad \land\ running\_task \neq idle \\
\quad \land\ topReady! \in state^{\sim}(\!|\ \{ready\}\ |\!) \\
\quad \land\ (\forall\, rtsk : state^{\sim}(\!|\ \{ready\}\ |\!) \\
\qquad \bullet\ priority(topReady!) \geq priority(rtsk)) \\
\quad \land\ wtime? > clock \\
\quad \land\ (running\_task \in \operatorname{dom} release\_rcv \\
\qquad \Rightarrow running\_task \in \operatorname{dom} release\_mutex) \\
\quad \land\ mut? \in \operatorname{dom} mutex\_holder \\
\quad \land\ q\_size(mut?) = 0 \\
\quad \land\ priority(running\_task) \leq priority(mutex\_holder(mut?)) \\
\quad \land\ running\_task \neq mutex\_holder(mut?) \\
\quad \bullet\ \neg\ (TaskQueueTimeMutex[log\_context := log\_context \oplus \\
\qquad\qquad \{(running\_task, phys\_context)\}, \\
\qquad\quad phys\_context := log\_context(topReady!), \\
\qquad\quad release\_mutex := \{running\_task\} \lhd release\_mutex, \\
\qquad\quad release\_rcv := \{running\_task\} \lhd release\_rcv, \\
\qquad\quad running\_task := topReady!, \\
\qquad\quad state := state \oplus \\
\qquad\qquad (\{(running\_task, blocked)\} \cup \{(topReady!, running)\}), \\
\qquad\quad wait\_time := wait\_time \oplus \{(running\_task, wtime?)\}, \\
\qquad\quad wait\_rcv := wait\_rcv \oplus \{(running\_task, mut?)\}] \\
\qquad \land\ (st \in TASK \\
\qquad\quad \land\ \neg\ (state \oplus (\{(running\_task, blocked)\} \cup \\
\qquad\qquad\qquad \{(topReady!, running)\}))st = state(st) \\
\qquad\quad \Rightarrow (state(st), (state \oplus (\{(running\_task, blocked)\} \cup \\
\qquad\qquad\qquad \{(topReady!, running)\}))st) \in transition) \\
\qquad \Rightarrow t \in TASK \\
\qquad\quad \land\ state(t) = ready \\
\qquad\quad \land\ \neg\ priority(topReady!) \geq priority(t))
\end{array}
$$

theorem MutexTakeEnonInh_TQTM_vc_ref

$$
\begin{array}{l}
\forall\, MutexTakeEnonInh\_TQTMFSBSig \mid true \\
\quad \bullet\ \text{pre}\ MutexTakeEnonInh\_TQTM
\end{array}
$$

MutexTakeEInheritReady_TQTM

$$
\begin{array}{l}
\Delta\, TaskQueueTimeMutex \\
mut? : QUEUE
\end{array}
$$

$$
\begin{array}{l}
topReady! : TASK \\
wtime? : \mathbb{N} \\
\hline
running\_task \notin \mathrm{dom}\, release\_snd \\
running\_task \in \mathrm{dom}\, release\_rcv \Rightarrow mut? = release\_rcv(running\_task) \\
running\_task \in \mathrm{dom}\, release\_rcv \Rightarrow running\_task \in \mathrm{dom}\, release\_mutex \\
mut? \in \mathrm{dom}\, mutex\_holder \\
priority(running\_task) > priority(mutex\_holder(mut?)) \\
wtime? > clock \\
mutex\_holder(mut?) \notin state^{\sim}(\!| \{ready\} |\!) \\
state(topReady!) = ready \\
\forall\, rt : state^{\sim}(\!| \{ready\} |\!) \bullet priority(topReady!) \geq priority(rt) \\
\exists\, st? : STATE;\ pri? : TASK \to \mathbb{N} \\
\quad | \ st? = blocked \land pri? = priority \oplus \\
\qquad \{(mutex\_holder(mut?) \mapsto priority(running\_task))\} \\
\quad \bullet\ Reschedule[topReady!/target?, tasks/tasks?] \\
\Xi QueueData \\
wait\_snd' = wait\_snd \\
wait\_rcv' = wait\_rcv \oplus \{(running\_task \mapsto mut?)\} \\
release\_snd' = release\_snd \\
release\_rcv' = \{running\_task\} \lhd release\_rcv \\
clock' = clock \\
delayed\_task' = delayed\_task \\
wait\_time' = wait\_time \oplus \{(running\_task \mapsto wtime?)\} \\
time\_slice' = time\_slice \\
\Xi MutexData \\
\Xi OriginalPrioData \\
release\_mutex' = \{running\_task\} \lhd release\_mutex
\end{array}
$$

MutexTakeEInheritReady_TQTMFSBSig

$$
\begin{array}{l}
TaskQueueTimeMutex \\
mut? : QUEUE \\
wtime? : \mathbb{N} \\
\hline
running\_task \notin \mathrm{dom}\, release\_snd \\
running\_task \in \mathrm{dom}\, release\_rcv \Rightarrow mut? = release\_rcv(running\_task) \\
running\_task \in \mathrm{dom}\, release\_rcv \Rightarrow running\_task \in \mathrm{dom}\, release\_mutex \\
mut? \in \mathrm{dom}\, mutex\_holder \\
priority(running\_task) > priority(mutex\_holder(mut?)) \\
wtime? > clock \\
mutex\_holder(mut?) \notin state^{\sim}(\!| \{ready\} |\!)
\end{array}
$$

theorem lMutexTakeEInheritReady_TQTM_Lemma

$$
\forall\, TaskQueueTimeMutex;\ mut? : QUEUE;\ topReady! : TASK;\ wtime? : \mathbb{N}
$$

$$
\begin{array}{l}
\mid running\_task \notin \mathrm{dom}\, release\_snd \\
\land\ (running\_task \in \mathrm{dom}\, release\_rcv \\
\quad \Rightarrow mut? = release\_rcv(running\_task)) \\
\land\ (running\_task \in \mathrm{dom}\, release\_rcv \\
\quad \Rightarrow running\_task \in \mathrm{dom}\, release\_mutex) \\
\land\ mut? \in \mathrm{dom}\, mutex\_holder \\
\land\ priority(running\_task) > priority(mutex\_holder(mut?)) \\
\land\ wtime? > clock \\
\land\ mutex\_holder(mut?) \notin state^{\sim}(\!|\ \{ready\}\ |\!) \\
\land\ state(topReady!) = ready \\
\land\ (\forall\, rtsk : state^{\sim}(\!|\ \{ready\}\ |\!) \\
\quad \bullet\ priority(topReady!) \geq priority(rtsk)) \\
\bullet\ \neg\ (TaskQueueTimeMutex[log\_context := log\_context \oplus \\
\qquad \{(running\_task, phys\_context)\}, \\
\qquad phys\_context := log\_context(topReady!), \\
\qquad priority := priority \oplus \\
\qquad\quad \{(mutex\_holder(mut?), priority(running\_task))\}, \\
\qquad release\_mutex := \{running\_task\} \lhd release\_mutex, \\
\qquad release\_rcv := \{running\_task\} \lhd release\_rcv, \\
\qquad running\_task := topReady!, \\
\qquad state := state \oplus \\
\qquad\quad (\{(running\_task, blocked)\} \cup \{(topReady!, running)\}), \\
\qquad wait\_time := wait\_time \oplus \{(running\_task, wtime?)\}, \\
\qquad wait\_rcv := wait\_rcv \oplus \{(running\_task, mut?)\}] \\
\quad \land\ (st \in TASK \\
\qquad \land\ \neg\ (state \oplus (\{(running\_task, blocked)\} \cup \\
\qquad\quad \{(topReady!, running)\}))st = state(st) \\
\qquad\quad \Rightarrow (state(st), (state \oplus (\{(running\_task, blocked)\} \cup \\
\qquad\quad\quad \{(topReady!, running)\}))st) \in transition) \\
\quad \Rightarrow rt \in TASK \\
\qquad \land\ state(rt) = ready \\
\qquad \land\ \neg\ priority(topReady!) \geq priority(rt))
\end{array}
$$

theorem MutexTakeEInheritReady_TQTM_vc_ref

$$
\begin{array}{l}
\forall\, MutexTakeEInheritReady\_TQTMFSBSig \mid true \\
\quad \bullet\ \mathrm{pre}\ MutexTakeEInheritReady\_TQTM
\end{array}
$$

MutexTakeEInheritHolder_TQTM

$$
\begin{array}{l}
\Delta\, TaskQueueTimeMutex \\
mut? : QUEUE \\
topReady! : TASK \\
wtime? : \mathbb{N} \\
\hline
running\_task \notin \mathrm{dom}\, release\_snd \\
running\_task \in \mathrm{dom}\, release\_rcv \Rightarrow mut? = release\_rcv(running\_task)
\end{array}
$$

$running\_task \in \mathrm{dom}\, release\_rcv \Rightarrow running\_task \in \mathrm{dom}\, release\_mutex$
$mut? \in \mathrm{dom}\, mutex\_holder$
$priority(running\_task) > priority(mutex\_holder(mut?))$
$wtime? > clock$
$mutex\_holder(mut?) \in state^{\sim}(\!|\ \{ready\}\ |\!)$
$topReady! = mutex\_holder(mut?)$
$topReady! \neq idle$
$\exists\, st? : STATE;\ pri? : TASK \rightarrow \mathbb{N}$
$\quad \mid st? = blocked \land pri? = priority \oplus$
$\quad\quad \{(topReady! \mapsto priority(running\_task))\}$
$\quad \bullet\ Reschedule[topReady!/target?, tasks/tasks?]$
$\Xi QueueData$
$wait\_snd' = wait\_snd$
$wait\_rcv' = wait\_rcv \oplus \{(running\_task \mapsto mut?)\}$
$release\_snd' = release\_snd$
$release\_rcv' = \{running\_task\} \lhd release\_rcv$
$clock' = clock$
$delayed\_task' = delayed\_task$
$wait\_time' = wait\_time \oplus \{(running\_task \mapsto wtime?)\}$
$time\_slice' = time\_slice$
$\Xi MutexData$
$\Xi OriginalPrioData$
$release\_mutex' = \{running\_task\} \lhd release\_mutex$

MutexTakeEInheritHolder_TQTMFSBSig

$TaskQueueTimeMutex$
$mut? : QUEUE$
$wtime? : \mathbb{N}$

$running\_task \notin \mathrm{dom}\, release\_snd$
$running\_task \in \mathrm{dom}\, release\_rcv \Rightarrow mut? = release\_rcv(running\_task)$
$running\_task \in \mathrm{dom}\, release\_rcv \Rightarrow running\_task \in \mathrm{dom}\, release\_mutex$
$mut? \in \mathrm{dom}\, mutex\_holder$
$priority(running\_task) > priority(mutex\_holder(mut?))$
$wtime? > clock$
$mutex\_holder(mut?) \neq idle$
$mutex\_holder(mut?) \in state^{\sim}(\!|\ \{ready\}\ |\!)$

theorem MutexTakeEInheritHolder_TQTM_vc_ref
$\forall\, MutexTakeEInheritHolder\_TQTMFSBSig \mid true$
$\quad \bullet$ pre $MutexTakeEInheritHolder\_TQTM$

MutexTakeRecursive_TQTM

$\Delta TaskQueueTimeMutex$

$mut? : QUEUE$
$topReady! : TASK$

$running\_task \notin \mathrm{dom}\ release\_snd \cup \mathrm{dom}\ release\_rcv$
$mut? \in \mathrm{dom}\ mutex\_holder$
$running\_task = mutex\_holder(mut?)$
$\Xi TaskQueueTime$
$semaphore' = semaphore$
$mutex' = mutex$
$mutex\_holder' = mutex\_holder$
$mutex\_recursive' = mutex\_recursive \oplus$
$\quad \{(mut? \mapsto mutex\_recursive(mut?) + 1)\}$
$\Xi OriginalPrioData$
$\Xi MReleasingData$
$topReady! = running\_task$

MutexTakeRecursive_TQTMFSBSig

$TaskQueueTimeMutex$
$mut? : QUEUE$

$running\_task \notin \mathrm{dom}\ release\_snd \cup \mathrm{dom}\ release\_rcv$
$mut? \in \mathrm{dom}\ mutex\_holder$
$running\_task = mutex\_holder(mut?)$

---

theorem MutexTakeRecursive_TQTM_vc_ref

$\forall MutexTakeRecursive\_TQTMFSBSig \mid true$
$\quad \bullet$ pre $MutexTakeRecursive\_TQTM$

$$
\begin{aligned}
MutexTake\_TQTM \mathrel{\widehat=}\ & MutexTakeNnonInh\_TQTM \\
& \lor\ MutexTakeNInh\_TQTM \\
& \lor\ MutexTakeEnonInh\_TQTM \\
& \lor\ MutexTakeEInheritReady\_TQTM \\
& \lor\ MutexTakeEInheritHolder\_TQTM \\
& \lor\ MutexTakeRecursive\_TQTM
\end{aligned}
$$

basePriorityMan

$\Delta TaskQueueTimeMutex$
$mut? : QUEUE$

$running\_task \in \mathrm{ran}(\{mut?\} \lhd mutex\_holder) \Rightarrow \Xi OriginalPrioData$
$running\_task \notin \mathrm{ran}(\{mut?\} \lhd mutex\_holder)$
$\quad \Rightarrow base\_priority' = \{running\_task\} \lhd base\_priority$

MutexGiveNnonInh_TQTM

$\Delta TaskQueueTimeMutex$
$mut? : QUEUE$
$topReady! : TASK$

$running\_task \notin \mathrm{dom}\, release\_snd \cup \mathrm{dom}\, release\_rcv$
$mut? \in \mathrm{dom}\, mutex\_holder$
$running\_task = mutex\_holder(mut?)$
$mutex\_recursive(mut?) = 1$
$base\_priority(running\_task) = priority(running\_task)$
$QueueSendN\_TQT[que? := mut?]$
$semaphore' = semaphore$
$mutex' = mutex$
$mutex\_holder' = \{mut?\} \lhd mutex\_holder$
$mutex\_recursive' = mutex\_recursive \oplus \{(mut? \mapsto 0)\}$
$basePriorityMan$
$\Xi MReleasingData$

MutexGiveNnonInh_TQTMFSBSig

$TaskQueueTimeMutex$
$mut? : QUEUE$

$QueueSendN\_TQTFSBSig[que? := mut?]$
$running\_task \notin \mathrm{dom}\, release\_snd \cup \mathrm{dom}\, release\_rcv$
$mut? \in \mathrm{dom}\, mutex\_holder$
$running\_task = mutex\_holder(mut?)$
$mutex\_recursive(mut?) = 1$
$base\_priority(running\_task) = priority(running\_task)$

theorem ranUnchanged $[X, Y]$

$\forall f : X \pfun Y;\ a : X \mid a \in \mathrm{dom}\, f \land f(a) \in \mathrm{ran}(\{a\} \lhd f)$
• $\mathrm{ran}\, f = \mathrm{ran}(\{a\} \lhd f)$

theorem MutexGiveNnonInh_TQTM_vc_ref

$\forall MutexGiveNnonInh\_TQTMFSBSig \mid true$
• pre $MutexGiveNnonInh\_TQTM$

MutexGiveNInhN_TQTM

$\Delta TaskQueueTimeMutex$
$mut? : QUEUE$
$topReady! : TASK$

$running\_task \notin \mathrm{dom}\, release\_snd \cup \mathrm{dom}\, release\_rcv$
$mut? \in \mathrm{dom}\, mutex\_holder$

$running\_task = mutex\_holder(mut?)$
$mutex\_recursive(mut?) = 1$
$mut? \notin \operatorname{ran} wait\_rcv$
$base\_priority(running\_task) \neq priority(running\_task)$
$\forall rt : state^{\sim}(\!| \{ready\} |\!) \bullet base\_priority(running\_task) \geq priority(rt)$
$\Xi TaskData$
$\Xi StateData$
$\Xi ContextData$
$priority' = priority \oplus \{(running\_task \mapsto base\_priority(running\_task))\}$
$queue' = queue$
$q\_max' = q\_max$
$q\_size' = q\_size \oplus \{(mut? \mapsto 1)\}$
$\Xi WaitingData$
$\Xi QReleasingData$
$\Xi Time$
$semaphore' = semaphore$
$mutex' = mutex$
$mutex\_holder' = \{mut?\} \lhd mutex\_holder$
$mutex\_recursive' = mutex\_recursive \oplus \{(mut? \mapsto 0)\}$
$basePriorityMan$
$\Xi MReleasingData$
$topReady! = running\_task$

MutexGiveNInhN_TQTMFSBSig

$TaskQueueTimeMutex$
$mut? : QUEUE$

$running\_task \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv$
$mut? \in \operatorname{dom} mutex\_holder$
$running\_task = mutex\_holder(mut?)$
$mutex\_recursive(mut?) = 1$
$mut? \notin \operatorname{ran} wait\_rcv$
$base\_priority(running\_task) \neq priority(running\_task)$
$\forall rt : state^{\sim}(\!| \{ready\} |\!) \bullet base\_priority(running\_task) \geq priority(rt)$

theorem MutexGiveNInhN_TQTMF_vc_ref
$\forall MutexGiveNInhN\_TQTMFSBSig \mid true$
• pre $MutexGiveNInhN\_TQTM$

MutexGiveNInhS_TQTM

$\Delta TaskQueueTimeMutex$
$mut? : QUEUE$
$topReady! : TASK$

$$
\begin{array}{l}
running\_task \notin \mathrm{dom}\, release\_snd \cup \mathrm{dom}\, release\_rcv \\
mut? \in \mathrm{dom}\, mutex\_holder \\
running\_task = mutex\_holder(mut?) \\
mutex\_recursive(mut?) = 1 \\
mut? \notin \mathrm{ran}\, wait\_rcv \\
base\_priority(running\_task) \neq priority(running\_task) \\
state(topReady!) = ready \\
\forall\, rt : state^{\sim}(\!|\, \{ready\} \,|\!) \bullet priority(topReady!) \geq priority(rt) \\
base\_priority(running\_task) < priority(topReady!) \\
\exists\, st? : STATE;\ pri? : TASK \to \mathbb{N} \\
\quad | \ st? = ready \land pri? = priority \oplus \\
\qquad \{(running\_task \mapsto base\_priority(running\_task))\} \\
\quad \bullet \ Reschedule[topReady!/target?, tasks/tasks?] \\
queue' = queue \\
q\_max' = q\_max \\
q\_size' = q\_size \oplus \{(mut? \mapsto 1)\} \\
\Xi WaitingData \\
\Xi QReleasingData \\
\Xi Time \\
semaphore' = semaphore \\
mutex' = mutex \\
mutex\_holder' = \{mut?\} \lhd mutex\_holder \\
mutex\_recursive' = mutex\_recursive \oplus \{(mut? \mapsto 0)\} \\
basePriorityMan \\
\Xi MReleasingData
\end{array}
$$

MutexGiveNInhS_TQTMFSBSig

$$
\begin{array}{l}
TaskQueueTimeMutex \\
mut? : QUEUE \\
\hline
running\_task \notin \mathrm{dom}\, release\_snd \cup \mathrm{dom}\, release\_rcv \\
mut? \in \mathrm{dom}\, mutex\_holder \\
running\_task = mutex\_holder(mut?) \\
mutex\_recursive(mut?) = 1 \\
mut? \notin \mathrm{ran}\, wait\_rcv \\
base\_priority(running\_task) \neq priority(running\_task) \\
\exists\, topReady! : state^{\sim}(\!|\, \{ready\} \,|\!) \\
\quad | \ \forall\, rt : state^{\sim}(\!|\, \{ready\} \,|\!) \\
\qquad \bullet \ priority(topReady!) \geq priority(rt) \\
\quad \bullet \ base\_priority(running\_task) < priority(topReady!)
\end{array}
$$

theorem lMutexGiveNInhS_TQTM_Lemma

$$
\begin{array}{l}
\forall\, TaskQueueTimeMutex;\ mut? : QUEUE;\ topReady! : TASK \\
\quad | \ running\_task \notin \mathrm{dom}\, release\_snd \cup \mathrm{dom}\, release\_rcv \\
\quad \land \ mut? \in \mathrm{dom}\, mutex\_holder
\end{array}
$$

$$
\begin{array}{l}
\quad \wedge\ running\_task = mutex\_holder(mut?) \\
\quad \wedge\ mutex\_recursive(mut?) = 1 \\
\quad \wedge\ mut? \notin \operatorname{ran} wait\_rcv \\
\quad \wedge\ base\_priority(running\_task) \neq priority(running\_task) \\
\quad \wedge\ state(topReady!) = ready \\
\quad \wedge\ (\forall\, rtsk : state^{\sim}(\!|\ \{ready\}\ |\!) \\
\qquad \bullet\ priority(topReady!) \geq priority(rtsk)) \\
\quad \wedge\ base\_priority(running\_task) < priority(topReady!) \\
\quad \wedge\ running\_task \in \operatorname{ran}(\{mut?\} \lhd mutex\_holder) \\
\quad \bullet \neg\ (TaskQueueTimeMutex[log\_context := log\_context \oplus \\
\qquad\qquad \{(mutex\_holder(mut?), phys\_context)\}, \\
\qquad\quad mutex\_holder := \{mut?\} \lhd mutex\_holder, \\
\qquad\quad mutex\_recursive := mutex\_recursive \oplus \{(mut?, 0)\}, \\
\qquad\quad phys\_context := log\_context(topReady!), \\
\qquad\quad priority := priority \oplus \{(mutex\_holder(mut?), \\
\qquad\qquad base\_priority(mutex\_holder(mut?)))\}, \\
\qquad\quad q\_size := q\_size \oplus \{(mut?, 1)\}, \\
\qquad\quad running\_task := topReady!, \\
\qquad\quad state := state \oplus (\{(mutex\_holder(mut?), ready)\} \cup \\
\qquad\qquad \{(topReady!, running)\})] \\
\qquad \wedge\ (st \in TASK \\
\qquad\quad \wedge \neg\ (state \oplus (\{(mutex\_holder(mut?), ready)\} \cup \\
\qquad\qquad\quad \{(topReady!, running)\}))st = state(st) \\
\qquad\quad \Rightarrow (state(st), (state \oplus \\
\qquad\qquad\quad (\{(mutex\_holder(mut?), ready)\} \cup \\
\qquad\qquad\qquad \{(topReady!, running)\}))st) \\
\qquad\qquad\qquad\quad \in transition) \\
\qquad \Rightarrow rt \in TASK \\
\qquad\quad \wedge\ state(rt) = ready \\
\qquad\quad \wedge \neg\ priority(topReady!) \geq priority(rt))
\end{array}
$$

theorem lMutexGiveNInhS_TQTM_Lemma1

$$
\begin{array}{l}
\quad \forall\, TaskQueueTimeMutex;\ mut? : QUEUE;\ topReady! : TASK \\
\qquad |\ running\_task \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv \\
\qquad \wedge\ mut? \in \operatorname{dom} mutex\_holder \\
\qquad \wedge\ running\_task = mutex\_holder(mut?) \\
\qquad \wedge\ mutex\_recursive(mut?) = 1 \\
\qquad \wedge\ mut? \notin \operatorname{ran} wait\_rcv \\
\qquad \wedge\ base\_priority(running\_task) \neq priority(running\_task) \\
\qquad \wedge\ state(topReady!) = ready \\
\qquad \wedge\ (\forall\, rtsk : state^{\sim}(\!|\ \{ready\}\ |\!) \\
\qquad\quad \bullet\ priority(topReady!) \geq priority(rtsk)) \\
\qquad \wedge\ base\_priority(running\_task) < priority(topReady!) \\
\qquad \wedge\ running\_task \notin \operatorname{ran}(\{mut?\} \lhd mutex\_holder) \\
\qquad \bullet \neg\ (TaskQueueTimeMutex[\\
\qquad\qquad\quad base\_priority := \{mutex\_holder(mut?)\} \lhd base\_priority,
\end{array}
$$

$$
\begin{array}{l}
\quad log\_context := log\_context \oplus \\
\qquad \{(mutex\_holder(mut?), phys\_context)\}, \\
\quad mutex\_holder := \{mut?\} \lhd mutex\_holder, \\
\quad mutex\_recursive := mutex\_recursive \oplus \{(mut?, 0)\}, \\
\quad phys\_context := log\_context(topReady!), \\
\quad priority := priority \oplus \{(mutex\_holder(mut?), \\
\qquad base\_priority(mutex\_holder(mut?)))\}, \\
\quad q\_size := q\_size \oplus \{(mut?, 1)\}, \\
\quad running\_task := topReady!, \\
\quad state := state \oplus (\{(mutex\_holder(mut?), ready)\} \cup \\
\qquad \{(topReady!, running)\})] \\
\land (st \in TASK \\
\quad \land \neg (state \oplus (\{(mutex\_holder(mut?), ready)\} \cup \\
\qquad \{(topReady!, running)\}))st = state(st) \\
\quad \Rightarrow (state(st), (state \oplus \\
\qquad (\{(mutex\_holder(mut?), ready)\} \cup \\
\qquad \{(topReady!, running)\}))st) \in transition) \\
\Rightarrow rt \in TASK \\
\quad \land state(rt) = ready \\
\quad \land \neg priority(topReady!) \geq priority(rt))
\end{array}
$$

theorem MutexGiveNInhS_TQTM_vc_ref

$\forall MutexGiveNInhS\_TQTMFSBSig \mid true \bullet \text{pre } MutexGiveNInhS\_TQTM$

MutexGiveWnonInhN_TQTM

$\Delta TaskQueueTimeMutex$
$mut? : QUEUE$
$topReady! : TASK$

$running\_task \notin \text{dom } release\_snd \cup \text{dom } release\_rcv$
$mut? \in \text{dom } mutex\_holder$
$running\_task = mutex\_holder(mut?)$
$mutex\_recursive(mut?) = 1$
$base\_priority(running\_task) = priority(running\_task)$
$QueueSendW\_TQT[que? := mut?]$
$semaphore' = semaphore$
$mutex' = mutex$
$mutex\_holder' = \{mut?\} \lhd mutex\_holder$
$mutex\_recursive' = mutex\_recursive \oplus \{(mut? \mapsto 0)\}$
$basePriorityMan$
$release\_mutex' = release\_mutex \oplus \{(topReady! \mapsto mut?)\}$

MutexGiveWnonInhN_TQTMFSBSig

$TaskQueueTimeMutex$

$mut? : QUEUE$

$QueueSendW\_TQTFSBSig[que? := mut?]$
$running\_task \notin \mathrm{dom}\, release\_snd \cup \mathrm{dom}\, release\_rcv$
$mut? \in \mathrm{dom}\, mutex\_holder$
$running\_task = mutex\_holder(mut?)$
$mutex\_recursive(mut?) = 1$
$base\_priority(running\_task) = priority(running\_task)$

theorem lMutexGiveWnonInhN_TQTM_Lemma

$\forall\ TaskQueueTimeMutex;\ topReady! : TASK;\ mut? : QUEUE$
$\mid running\_task \notin \mathrm{dom}\, release\_rcv \cup \mathrm{dom}\, release\_snd$
$\land\ mut? \in queue$
$\land\ q\_size(mut?) < q\_max(mut?)$
$\land\ topReady! \in wait\_rcv^{\sim}(\!|\ \{mut?\}\ |\!)$
$\land\ (\forall\ wrct : wait\_rcv^{\sim}(\!|\ \{mut?\}\ |\!)$
$\bullet\ priority(topReady!) \geq priority(wrct))$
$\land\ priority(running\_task) \geq priority(topReady!)$
$\land\ mut? \in \mathrm{dom}\, mutex\_holder$
$\land\ running\_task = mutex\_holder(mut?)$
$\land\ mutex\_recursive(mut?) = 1$
$\land\ base\_priority(running\_task) = priority(running\_task)$
$\land\ running\_task \in \mathrm{ran}(\{mut?\} \lhd mutex\_holder)$
$\bullet \neg\ (TaskQueueTimeMutex[$
$mutex\_holder := \{mut?\} \lhd mutex\_holder,$
$mutex\_recursive := mutex\_recursive \oplus \{(mut?, 0)\},$
$q\_size := q\_size \oplus \{(mut?, (1 + q\_size(mut?)))\},$
$release\_mutex := release\_mutex \oplus \{(topReady!, mut?)\},$
$release\_rcv := release\_rcv \oplus \{(topReady!, mut?)\},$
$release\_snd := \{mutex\_holder(mut?)\} \lhd release\_snd,$
$running\_task := mutex\_holder(mut?),$
$state := state \oplus \{(topReady!, ready)\},$
$wait\_time := \{topReady!\} \lhd wait\_time,$
$wait\_rcv := \{topReady!\} \lhd wait\_rcv]$
$\land\ priority(topReady!) \leq priority(mutex\_holder(mut?))$
$\land\ (st \in TASK$
$\land \neg\ (state \oplus \{(topReady!, ready)\})st = state(st)$
$\Rightarrow (state(st), (state \oplus \{(topReady!, ready)\})st)$
$\in transition)$
$\Rightarrow wr \in \mathrm{dom}\, wait\_rcv$
$\land\ wait\_rcv(wr) = mut?$
$\land \neg\ priority(topReady!) \geq priority(wr))$

theorem lMutexGiveWnonInhN_TQTM_Lemma1

$\forall\ TaskQueueTimeMutex;\ topReady! : TASK;\ mut? : QUEUE$

$$
\begin{array}{l}
\mid running\_task \notin \mathrm{dom}\, release\_rcv \cup \mathrm{dom}\, release\_snd \\
\land\ mut? \in queue \\
\land\ q\_size(mut?) < q\_max(mut?) \\
\land\ topReady! \in wait\_rcv^{\sim}(\!|\ \{mut?\}\ |\!) \\
\land\ (\forall\, wrct : wait\_rcv^{\sim}(\!|\ \{mut?\}\ |\!) \\
\quad \bullet\ priority(topReady!) \geq priority(wrct)) \\
\land\ priority(running\_task) \geq priority(topReady!) \\
\land\ mut? \in \mathrm{dom}\, mutex\_holder \\
\land\ running\_task = mutex\_holder(mut?) \\
\land\ mutex\_recursive(mut?) = 1 \\
\land\ base\_priority(running\_task) = priority(running\_task) \\
\land\ running\_task \notin \mathrm{ran}(\{mut?\} \lhd mutex\_holder) \\
\bullet\ \neg\ (TaskQueueTimeMutex[\\
\qquad base\_priority := \{mutex\_holder(mut?)\} \lhd base\_priority, \\
\qquad mutex\_holder := \{mut?\} \lhd mutex\_holder, \\
\qquad mutex\_recursive := mutex\_recursive \oplus \{(mut?, 0)\}, \\
\qquad q\_size := q\_size \oplus \{(mut?, (1 + q\_size(mut?)))\}, \\
\qquad release\_mutex := release\_mutex \oplus \{(topReady!, mut?)\}, \\
\qquad release\_rcv := release\_rcv \oplus \{(topReady!, mut?)\}, \\
\qquad release\_snd := \{mutex\_holder(mut?)\} \lhd release\_snd, \\
\qquad running\_task := mutex\_holder(mut?), \\
\qquad state := state \oplus \{(topReady!, ready)\}, \\
\qquad wait\_time := \{topReady!\} \lhd wait\_time, \\
\qquad wait\_rcv := \{topReady!\} \lhd wait\_rcv] \\
\quad \land\ priority(topReady!) \leq priority(mutex\_holder(mut?)) \\
\quad \land\ (st \in TASK \\
\qquad \land\ \neg\ (state \oplus \{(topReady!, ready)\})st = state(st) \\
\qquad\quad \Rightarrow (state(st), (state \oplus \{(topReady!, ready)\})st) \\
\qquad\qquad \in transition) \\
\quad \Rightarrow wr \in \mathrm{dom}\, wait\_rcv \\
\qquad \land\ wait\_rcv(wr) = mut? \\
\qquad \land\ \neg\ priority(topReady!) \geq priority(wr))
\end{array}
$$

theorem MutexGiveWnonInhN_TQTM_vc_ref

$$
\forall\, MutexGiveWnonInhN\_TQTMFSBSig \mid true \bullet \text{pre } MutexGiveWnonInhN\_TQTM
$$

MutexGiveWnonInhS_TQTM

$$
\begin{array}{l}
\Delta TaskQueueTimeMutex \\
mut? : QUEUE \\
topReady! : TASK \\
\hline
running\_task \notin \mathrm{dom}\, release\_snd \cup \mathrm{dom}\, release\_rcv \\
mut? \in \mathrm{dom}\, mutex\_holder \\
running\_task = mutex\_holder(mut?)
\end{array}
$$

$mutex\_recursive(mut?) = 1$
$base\_priority(running\_task) = priority(running\_task)$
$QueueSendWS\_TQT[que? := mut?]$
$semaphore' = semaphore$
$mutex' = mutex$
$mutex\_holder' = \{mut?\} \lhd mutex\_holder$
$mutex\_recursive' = mutex\_recursive \oplus \{(mut? \mapsto 0)\}$
$basePriorityMan$
$release\_mutex' = release\_mutex \oplus \{(topReady! \mapsto mut?)\}$

MutexGiveWnonInhS_TQTMFSBSig

$TaskQueueTimeMutex$
$mut? : QUEUE$

$QueueSendWS\_TQTFSBSig[que? := mut?]$
$running\_task \notin \mathrm{dom}\, release\_snd \cup \mathrm{dom}\, release\_rcv$
$mut? \in \mathrm{dom}\, mutex\_holder$
$running\_task = mutex\_holder(mut?)$
$mutex\_recursive(mut?) = 1$
$base\_priority(running\_task) = priority(running\_task)$

theorem lMutexGiveWnonInhS_TQTM_Lemma

$$
\begin{array}{l}
\forall\ TaskQueueTimeMutex;\ topReady! : TASK;\ mut? : QUEUE \\
\quad \mid running\_task \notin \mathrm{dom}\, release\_rcv \\
\quad \land\ (running\_task \in \mathrm{dom}\, release\_snd \\
\qquad \Rightarrow mut? = release\_snd(running\_task)) \\
\quad \land\ mut? \in queue \\
\quad \land\ q\_size(mut?) < q\_max(mut?) \\
\quad \land\ topReady! \in wait\_rcv^{\sim}(\!|\ \{mut?\}\ |\!) \\
\quad \land\ (\forall\ wrct : wait\_rcv^{\sim}(\!|\ \{mut?\}\ |\!) \\
\qquad \bullet\ priority(topReady!) \geq priority(wrct)) \\
\quad \land\ priority(topReady!) > priority(running\_task) \\
\quad \land\ running\_task \notin \mathrm{dom}\, release\_snd \cup \mathrm{dom}\, release\_rcv \\
\quad \land\ mut? \in \mathrm{dom}\, mutex\_holder \\
\quad \land\ running\_task = mutex\_holder(mut?) \\
\quad \land\ mutex\_recursive(mut?) = 1 \\
\quad \land\ base\_priority(running\_task) = priority(running\_task) \\
\quad \land\ running\_task \in \mathrm{ran}(\{mut?\} \lhd mutex\_holder) \\
\quad \bullet\ \neg\ (TaskQueueTimeMutex[log\_context := log\_context \oplus \\
\qquad\qquad \{(mutex\_holder(wait\_rcv(topReady!)), phys\_context)\}, \\
\qquad mutex\_holder := \{wait\_rcv(topReady!)\} \lhd mutex\_holder, \\
\qquad mutex\_recursive := mutex\_recursive \oplus \\
\qquad\qquad \{(wait\_rcv(topReady!), 0)\}, \\
\qquad phys\_context := log\_context(topReady!), \\
\qquad q\_size := q\_size \oplus \{(wait\_rcv(topReady!), (1+
\end{array}
$$

$$
\begin{array}{l}
\qquad q\_size(wait\_rcv(topReady!)))))\},\\
\quad release\_mutex := release\_mutex\oplus\\
\qquad \{(topReady!, wait\_rcv(topReady!))\},\\
\quad release\_rcv := release\_rcv\oplus\\
\qquad \{(topReady!, wait\_rcv(topReady!))\},\\
\quad release\_snd := \{mutex\_holder(wait\_rcv(topReady!))\}\\
\qquad \lhd release\_snd,\\
\quad running\_task := topReady!,\\
\quad state := state\oplus\\
\qquad (\{(mutex\_holder(wait\_rcv(topReady!)), ready)\}\cup\\
\qquad\quad \{(topReady!, running)\}),\\
\quad wait\_time := \{topReady!\} \lhd wait\_time,\\
\quad wait\_rcv := \{topReady!\} \lhd wait\_rcv]\\
\wedge\ (st \in TASK\\
\quad \wedge \neg\ (state\oplus\\
\qquad\quad (\{(mutex\_holder(wait\_rcv(topReady!)), ready)\}\cup\\
\qquad\qquad \{(topReady!, running)\}))st = state(st)\\
\qquad \Rightarrow (state(st), (state\oplus\\
\qquad\quad (\{(mutex\_holder(wait\_rcv(topReady!)), ready)\}\cup\\
\qquad\qquad \{(topReady!, running)\}))st) \in transition)\\
\Rightarrow\ wr \in \mathrm{dom}\ wait\_rcv\\
\quad \wedge\ wait\_rcv(wr) = wait\_rcv(topReady!)\\
\quad \wedge \neg\ priority(topReady!) \geq priority(wr))
\end{array}
$$

theorem lMutexGiveWnonInhS_TQTM_Lemma1

$$
\begin{array}{l}
\forall\ TaskQueueTimeMutex;\ topReady! : TASK;\ mut? : QUEUE\\
\quad \mid running\_task \notin \mathrm{dom}\ release\_rcv\\
\quad \wedge\ (running\_task \in \mathrm{dom}\ release\_snd\\
\qquad \Rightarrow mut? = release\_snd(running\_task))\\
\quad \wedge\ mut? \in queue\\
\quad \wedge\ q\_size(mut?) < q\_max(mut?)\\
\quad \wedge\ topReady! \in wait\_rcv^{\sim}(\!|\ \{mut?\}\ |\!)\\
\quad \wedge\ (\forall\ wrct : wait\_rcv^{\sim}(\!|\ \{mut?\}\ |\!)\\
\qquad \bullet\ priority(topReady!) \geq priority(wrct))\\
\quad \wedge\ priority(topReady!) > priority(running\_task)\\
\quad \wedge\ running\_task \notin \mathrm{dom}\ release\_snd \cup \mathrm{dom}\ release\_rcv\\
\quad \wedge\ mut? \in \mathrm{dom}\ mutex\_holder\\
\quad \wedge\ running\_task = mutex\_holder(mut?)\\
\quad \wedge\ mutex\_recursive(mut?) = 1\\
\quad \wedge\ base\_priority(running\_task) = priority(running\_task)\\
\quad \wedge\ running\_task \notin \mathrm{ran}(\{mut?\} \lhd mutex\_holder)\\
\quad \bullet \neg\ (TaskQueueTimeMutex[\\
\qquad\quad base\_priority := \{mutex\_holder(wait\_rcv(topReady!))\}\\
\qquad\qquad \lhd base\_priority,\\
\qquad\quad log\_context := log\_context\oplus\\
\qquad\qquad \{(mutex\_holder(wait\_rcv(topReady!)), phys\_context)\},
\end{array}
$$

$$
\begin{array}{l}
\quad mutex\_holder := \{wait\_rcv(topReady!)\} \lhd mutex\_holder, \\
\quad mutex\_recursive := mutex\_recursive \oplus \\
\qquad \{(wait\_rcv(topReady!), 0)\}, \\
\quad phys\_context := log\_context(topReady!), \\
\quad q\_size := q\_size \oplus \{(wait\_rcv(topReady!), (1 + \\
\qquad q\_size(wait\_rcv(topReady!)))\}, \\
\quad release\_mutex := release\_mutex \oplus \\
\qquad \{(topReady!, wait\_rcv(topReady!))\}, \\
\quad release\_rcv := release\_rcv \oplus \\
\qquad \{(topReady!, wait\_rcv(topReady!))\}, \\
\quad release\_snd := \{mutex\_holder(wait\_rcv(topReady!))\} \\
\qquad \lhd release\_snd, \\
\quad running\_task := topReady!, \\
\quad state := state \oplus \\
\qquad (\{(mutex\_holder(wait\_rcv(topReady!)), ready)\} \cup \\
\qquad\quad \{(topReady!, running)\}), \\
\quad wait\_time := \{topReady!\} \lhd wait\_time, \\
\quad wait\_rcv := \{topReady!\} \lhd wait\_rcv] \\
\land\ (st \in TASK \\
\quad \land \neg\ (state \oplus \\
\qquad\quad (\{(mutex\_holder(wait\_rcv(topReady!)), ready)\} \cup \\
\qquad\qquad \{(topReady!, running)\}))st = state(st) \\
\qquad \Rightarrow (state(st), (state \oplus \\
\qquad\quad (\{(mutex\_holder(wait\_rcv(topReady!)), ready)\} \cup \\
\qquad\qquad \{(topReady!, running)\}))st) \in transition) \\
\Rightarrow wr \in \mathrm{dom}\ wait\_rcv \\
\quad \land\ wait\_rcv(wr) = wait\_rcv(topReady!) \\
\quad \land \neg\ priority(topReady!) \geq priority(wr))
\end{array}
$$

theorem MutexGiveWnonInhS_TQTM_vc_ref

$\forall\ MutexGiveWnonInhS\_TQTMFSBSig \mid true$

• pre $MutexGiveWnonInhS\_TQTM$

MutexGiveWInhN_TQTM

$\Delta TaskQueueTimeMutex$

$mut? : QUEUE$

$topReady! : TASK$

$running\_task \notin \mathrm{dom}\ release\_snd \cup \mathrm{dom}\ release\_rcv$

$mut? \in \mathrm{dom}\ mutex\_holder$

$running\_task = mutex\_holder(mut?)$

$mutex\_recursive(mut?) = 1$

$topReady! \in wait\_rcv^{\sim}(\!|\ \{mut?\}\ |\!)$

$\forall\ wr : wait\_rcv^{\sim}(\!|\ \{mut?\}\ |\!) \bullet priority(topReady!) \geq priority(wr)$

$priority(topReady!) \leq base\_priority(running\_task)$

$base\_priority(running\_task) \neq priority(running\_task)$
$\forall\, rt : state^{\sim}(\!|\, \{ready\} \,|\!) \bullet base\_priority(running\_task) \geq priority(rt)$
$\Xi TaskData$
$state' = state \oplus \{(topReady! \mapsto ready)\}$
$\Xi ContextData$
$priority' = priority \oplus \{(running\_task \mapsto base\_priority(running\_task))\}$
$queue' = queue$
$q\_max' = q\_max$
$q\_size' = q\_size \oplus \{(mut? \mapsto 1)\}$
$wait\_snd' = wait\_snd$
$wait\_rcv' = \{topReady!\} \lhd wait\_rcv$
$release\_snd' = release\_snd$
$release\_rcv' = release\_rcv \oplus \{(topReady! \mapsto mut?)\}$
$clock' = clock$
$delayed\_task' = delayed\_task$
$wait\_time' = \{topReady!\} \lhd wait\_time$
$time\_slice' = time\_slice$
$semaphore' = semaphore$
$mutex' = mutex$
$mutex\_holder' = \{mut?\} \lhd mutex\_holder$
$mutex\_recursive' = mutex\_recursive \oplus \{(mut? \mapsto 0)\}$
$basePriorityMan$
$release\_mutex' = release\_mutex \oplus \{(topReady! \mapsto mut?)\}$

MutexGiveWInhN_TQTMFSBSig

$TaskQueueTimeMutex$
$mut? : QUEUE$

$running\_task \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv$
$mut? \in \operatorname{dom} mutex\_holder$
$running\_task = mutex\_holder(mut?)$
$mutex\_recursive(mut?) = 1$
$\forall\, wr : wait\_rcv^{\sim}(\!|\, \{mut?\} \,|\!)$
$\quad \bullet\ base\_priority(running\_task) \geq priority(wr)$
$base\_priority(running\_task) \neq priority(running\_task)$
$\forall\, rt : state^{\sim}(\!|\, \{ready\} \,|\!) \bullet base\_priority(running\_task) \geq priority(rt)$

theorem lMutexGiveWInhN_TQTM_Lemma
$\forall\, TaskQueueTimeMutex;\ topReady! : TASK;\ mut? : QUEUE$
$\quad | \ running\_task \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv$
$\quad \wedge\ mut? \in \operatorname{dom} mutex\_holder$
$\quad \wedge\ running\_task = mutex\_holder(mut?)$
$\quad \wedge\ mutex\_recursive(mut?) = 1$
$\quad \wedge\ topReady! \in wait\_rcv^{\sim}(\!|\, \{mut?\} \,|\!)$
$\quad \wedge\ (\forall\, wrct : wait\_rcv^{\sim}(\!|\, \{mut?\} \,|\!)$

$$
\begin{array}{l}
\quad \bullet\ priority(topReady!) \geq priority(wrct)) \\
\wedge\ priority(topReady!) \leq base\_priority(running\_task) \\
\wedge\ base\_priority(running\_task) \neq priority(running\_task) \\
\wedge\ (\forall\, rtsk : state^{\sim}(\!|\ \{ready\}\ |\!) \\
\bullet\ base\_priority(running\_task) \geq priority(rtsk)) \\
\wedge\ running\_task \in \mathrm{ran}(\{mut?\} \lhd mutex\_holder) \\
\bullet\ \neg\ (TaskQueueTimeMutex[\\
\qquad mutex\_holder := \{mut?\} \lhd mutex\_holder, \\
\qquad mutex\_recursive := mutex\_recursive \oplus \{(mut?, 0)\}, \\
\qquad priority := priority \oplus \{(mutex\_holder(mut?), \\
\qquad\quad base\_priority(mutex\_holder(mut?)))\}, \\
\qquad q\_size := q\_size \oplus \{(mut?, 1)\}, \\
\qquad release\_mutex := release\_mutex \oplus \{(topReady!, mut?)\}, \\
\qquad release\_rcv := release\_rcv \oplus \{(topReady!, mut?)\}, \\
\qquad running\_task := mutex\_holder(mut?), \\
\qquad state := state \oplus \{(topReady!, ready)\}, \\
\qquad wait\_time := \{topReady!\} \lhd wait\_time, \\
\qquad wait\_rcv := \{topReady!\} \lhd wait\_rcv] \\
\quad \wedge\ priority(topReady!) \leq base\_priority(mutex\_holder(mut?)) \\
\quad \wedge\ (st \in TASK \\
\qquad \wedge\ \neg\ (state \oplus \{(topReady!, ready)\})st = state(st) \\
\qquad\quad \Rightarrow (state(st), (state \oplus \{(topReady!, ready)\})st) \\
\qquad\qquad \in transition) \\
\quad \Rightarrow wr \in \mathrm{dom}\ wait\_rcv \\
\qquad \wedge\ wait\_rcv(wr) = mut? \\
\qquad \wedge\ \neg\ priority(topReady!) \geq priority(wr))
\end{array}
$$

theorem lMutexGiveWInhN_TQTM_Lemma1

$$
\begin{array}{l}
\forall\, TaskQueueTimeMutex;\ topReady! : TASK;\ mut? : QUEUE \\
\quad |\ running\_task \notin \mathrm{dom}\ release\_snd \cup \mathrm{dom}\ release\_rcv \\
\quad \wedge\ mut? \in \mathrm{dom}\ mutex\_holder \\
\quad \wedge\ running\_task = mutex\_holder(mut?) \\
\quad \wedge\ mutex\_recursive(mut?) = 1 \\
\quad \wedge\ topReady! \in wait\_rcv^{\sim}(\!|\ \{mut?\}\ |\!) \\
\quad \wedge\ (\forall\, wrct : wait\_rcv^{\sim}(\!|\ \{mut?\}\ |\!) \\
\qquad \bullet\ priority(topReady!) \geq priority(wrct)) \\
\quad \wedge\ priority(topReady!) \leq base\_priority(running\_task) \\
\quad \wedge\ base\_priority(running\_task) \neq priority(running\_task) \\
\quad \wedge\ (\forall\, rtsk : state^{\sim}(\!|\ \{ready\}\ |\!) \\
\qquad \bullet\ base\_priority(running\_task) \geq priority(rtsk)) \\
\quad \wedge\ running\_task \notin \mathrm{ran}(\{mut?\} \lhd mutex\_holder) \\
\quad \bullet\ \neg\ (TaskQueueTimeMutex[\\
\qquad base\_priority := \{mutex\_holder(mut?)\} \lhd base\_priority, \\
\qquad mutex\_holder := \{mut?\} \lhd mutex\_holder, \\
\qquad mutex\_recursive := mutex\_recursive \oplus \{(mut?, 0)\}, \\
\qquad priority := priority \oplus \{(mutex\_holder(mut?),
\end{array}
$$

$$
\begin{array}{l}
\qquad base\_priority(mutex\_holder(mut?)))\},\\
\quad q\_size := q\_size \oplus \{(mut?, 1)\},\\
\quad release\_mutex := release\_mutex \oplus \{(topReady!, mut?)\},\\
\quad release\_rcv := release\_rcv \oplus \{(topReady!, mut?)\},\\
\quad running\_task := mutex\_holder(mut?),\\
\quad state := state \oplus \{(topReady!, ready)\},\\
\quad wait\_time := \{topReady!\} \lhd wait\_time,\\
\quad wait\_rcv := \{topReady!\} \lhd wait\_rcv]\\
\land\ priority(topReady!) \leq base\_priority(mutex\_holder(mut?))\\
\land\ (st \in TASK\\
\quad \land \neg\ (state \oplus \{(topReady!, ready)\})st = state(st)\\
\qquad \Rightarrow (state(st), (state \oplus \{(topReady!, ready)\})st)\\
\qquad\quad \in transition)\\
\Rightarrow wr \in \mathrm{dom}\ wait\_rcv\\
\quad \land\ wait\_rcv(wr) = mut?\\
\quad \land \neg\ priority(topReady!) \geq priority(wr))
\end{array}
$$

theorem MutexGiveWInhN_TQTM_vc_ref

$$
\begin{array}{l}
\forall\ MutexGiveWInhN\_TQTMFSBSig \mid true\\
\quad \bullet\ \mathrm{pre}\ MutexGiveWInhN\_TQTM
\end{array}
$$

MutexGiveWInhSR_TQTM

$$
\begin{array}{l}
\Delta TaskQueueTimeMutex\\
mut? : QUEUE\\
topWaiting! : TASK\\
topReady! : TASK\\
\hline
running\_task \notin \mathrm{dom}\ release\_snd \cup \mathrm{dom}\ release\_rcv\\
mut? \in \mathrm{dom}\ mutex\_holder\\
running\_task = mutex\_holder(mut?)\\
mutex\_recursive(mut?) = 1\\
topWaiting! \in wait\_rcv^{\sim}(\!|\ \{mut?\}\ |\!)\\
\forall\ wr : wait\_rcv^{\sim}(\!|\ \{mut?\}\ |\!) \bullet priority(topWaiting!) \geq priority(wr)\\
state(topReady!) = ready\\
\forall\ rt : state^{\sim}(\!|\ \{ready\}\ |\!) \bullet priority(topReady!) \geq priority(rt)\\
base\_priority(running\_task) \neq priority(running\_task)\\
priority(topReady!) > priority(topWaiting!)\\
priority(topReady!) > base\_priority(running\_task)\\
tasks' = tasks\\
running\_task' = topReady!\\
state' = state \oplus \{(running\_task \mapsto ready),\\
\quad (topReady! \mapsto running), (topWaiting! \mapsto ready)\}\\
phys\_context' = log\_context(topReady!)\\
log\_context' = log\_context \oplus \{(running\_task \mapsto phys\_context)\}\\
priority' = priority \oplus \{(running\_task \mapsto base\_priority(running\_task))\}
\end{array}
$$

$$
\begin{array}{l}
queue' = queue \\
q\_max' = q\_max \\
q\_size' = q\_size \oplus \{(mut? \mapsto 1)\} \\
wait\_snd' = wait\_snd \\
wait\_rcv' = \{topWaiting!\} \lhd wait\_rcv \\
release\_snd' = release\_snd \\
release\_rcv' = release\_rcv \oplus \{(topWaiting! \mapsto mut?)\} \\
clock' = clock \\
delayed\_task' = delayed\_task \\
wait\_time' = \{topWaiting!\} \lhd wait\_time \\
time\_slice' = time\_slice \\
semaphore' = semaphore \\
mutex' = mutex \\
mutex\_holder' = \{mut?\} \lhd mutex\_holder \\
mutex\_recursive' = mutex\_recursive \oplus \{(mut? \mapsto 0)\} \\
basePriorityMan \\
release\_mutex' = release\_mutex \oplus \{(topWaiting! \mapsto mut?)\}
\end{array}
$$

MutexGiveWInhSR_TQTMFSBSig

$$
\begin{array}{l}
TaskQueueTimeMutex \\
mut? : QUEUE \\
\hline
running\_task \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv \\
mut? \in \operatorname{dom} mutex\_holder \\
running\_task = mutex\_holder(mut?) \\
mutex\_recursive(mut?) = 1 \\
base\_priority(running\_task) \neq priority(running\_task) \\
\exists\, topReady! : state^{\sim}(\!| \{ready\} |\!) \\
\quad \bullet\ (\forall\, rt : state^{\sim}(\!| \{ready\} |\!) \bullet priority(topReady!) \geq priority(rt)) \\
\qquad \wedge\ (\forall\, wr : wait\_rcv^{\sim}(\!| \{mut?\} |\!) \\
\qquad\quad \bullet\ priority(topReady!) > priority(wr)) \\
\qquad \wedge\ priority(topReady!) > base\_priority(running\_task)
\end{array}
$$

theorem lMutexGiveWInhSR_TQTM_Lemma

$$
\begin{array}{l}
\forall\, TaskQueueTimeMutex;\ topReady!, topWaiting! : TASK; \\
\quad mut? : QUEUE \\
\quad | \ running\_task \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv \\
\quad \wedge\ mut? \in \operatorname{dom} mutex\_holder \\
\quad \wedge\ running\_task = mutex\_holder(mut?) \\
\quad \wedge\ mutex\_recursive(mut?) = 1 \\
\quad \wedge\ topWaiting! \in wait\_rcv^{\sim}(\!| \{mut?\} |\!) \\
\quad \wedge\ (\forall\, wrct : wait\_rcv^{\sim}(\!| \{mut?\} |\!) \\
\qquad \bullet\ priority(topWaiting!) \geq priority(wrct)) \\
\quad \wedge\ state(topReady!) = ready \\
\quad \wedge\ (\forall\, rtsk : state^{\sim}(\!| \{ready\} |\!)
\end{array}
$$

$$
\begin{array}{l}
\quad \bullet\ priority(topReady!) \geq priority(rtsk)) \\
\land\ base\_priority(running\_task) \neq priority(running\_task) \\
\land\ priority(topReady!) > priority(topWaiting!) \\
\land\ priority(topReady!) > base\_priority(running\_task) \\
\land\ running\_task \in \mathrm{ran}(\{mut?\} \lhd mutex\_holder) \\
\bullet \neg\ (TaskQueueTimeMutex[log\_context := log\_context \oplus \\
\qquad \{(mutex\_holder(mut?), phys\_context)\}, \\
\qquad mutex\_holder := \{mut?\} \lhd mutex\_holder, \\
\qquad mutex\_recursive := mutex\_recursive \oplus \{(mut?, 0)\}, \\
\qquad phys\_context := log\_context(topReady!), \\
\qquad priority := priority \oplus \{(mutex\_holder(mut?), \\
\qquad\quad base\_priority(mutex\_holder(mut?)))\}, \\
\qquad q\_size := q\_size \oplus \{(mut?, 1)\}, \\
\qquad release\_mutex := release\_mutex \oplus \{(topWaiting!, mut?)\}, \\
\qquad release\_rcv := release\_rcv \oplus \{(topWaiting!, mut?)\}, \\
\qquad running\_task := topReady!, \\
\qquad state := state \oplus (\{(mutex\_holder(mut?), ready)\} \cup \\
\qquad\quad (\{(topReady!, running)\} \cup \{(topWaiting!, ready)\})), \\
\qquad wait\_time := \{topWaiting!\} \lhd wait\_time, \\
\qquad wait\_rcv := \{topWaiting!\} \lhd wait\_rcv] \\
\quad \land\ priority(topReady!) > priority(topWaiting!) \\
\quad \land\ (st \in TASK \\
\qquad \land \neg\ (state \oplus (\{(mutex\_holder(mut?), ready)\} \cup \\
\qquad\quad (\{(topReady!, running)\} \cup \\
\qquad\quad\quad \{(topWaiting!, ready)\})))st = state(st) \\
\qquad \Rightarrow (state(st), (state \oplus \\
\qquad\quad (\{(mutex\_holder(mut?), ready)\} \cup \\
\qquad\quad\quad (\{(topReady!, running)\} \cup \\
\qquad\quad\quad\quad \{(topWaiting!, ready)\})))st) \\
\qquad\quad\quad\quad \in transition) \\
\quad \land\ (wr \in \mathrm{dom}\ wait\_rcv \land wait\_rcv(wr) = mut? \\
\qquad \Rightarrow priority(topWaiting!) \geq priority(wr)) \\
\quad \Rightarrow rt \in TASK \\
\qquad \land\ state(rt) = ready \\
\qquad \land \neg\ priority(topReady!) \geq priority(rt))
\end{array}
$$

theorem lMutexGiveWInhSR_TQTM_Lemma1

$$
\begin{array}{l}
\forall\ TaskQueueTimeMutex;\ topReady!, topWaiting! : TASK; \\
\quad mut? : QUEUE \\
\quad |\ running\_task \notin \mathrm{dom}\ release\_snd \cup \mathrm{dom}\ release\_rcv \\
\quad \land\ mut? \in \mathrm{dom}\ mutex\_holder \\
\quad \land\ running\_task = mutex\_holder(mut?) \\
\quad \land\ mutex\_recursive(mut?) = 1 \\
\quad \land\ topWaiting! \in wait\_rcv^{\sim}(\!|\ \{mut?\}\ |\!) \\
\quad \land\ (\forall\ wrct : wait\_rcv^{\sim}(\!|\ \{mut?\}\ |\!) \\
\qquad \bullet\ priority(topWaiting!) \geq priority(wrct))
\end{array}
$$

$$
\begin{array}{l}
\quad \wedge\ state(topReady!) = ready \\
\quad \wedge\ (\forall\, rtsk : state^{\sim}(\!|\ \{ready\}\ |\!) \\
\qquad \bullet\ priority(topReady!) \geq priority(rtsk)) \\
\quad \wedge\ base\_priority(running\_task) \neq priority(running\_task) \\
\quad \wedge\ priority(topReady!) > priority(topWaiting!) \\
\quad \wedge\ priority(topReady!) > base\_priority(running\_task) \\
\quad \wedge\ running\_task \notin \mathrm{ran}(\{mut?\} \lhd mutex\_holder) \\
\quad \bullet\ \neg\ (TaskQueueTimeMutex[\\
\qquad\qquad base\_priority := \{mutex\_holder(mut?)\} \lhd base\_priority, \\
\qquad\qquad log\_context := log\_context \oplus \\
\qquad\qquad\qquad \{(mutex\_holder(mut?), phys\_context)\}, \\
\qquad\qquad mutex\_holder := \{mut?\} \lhd mutex\_holder, \\
\qquad\qquad mutex\_recursive := mutex\_recursive \oplus \{(mut?, 0)\}, \\
\qquad\qquad phys\_context := log\_context(topReady!), \\
\qquad\qquad priority := priority \oplus \{(mutex\_holder(mut?), \\
\qquad\qquad\qquad base\_priority(mutex\_holder(mut?)))\}, \\
\qquad\qquad q\_size := q\_size \oplus \{(mut?, 1)\}, \\
\qquad\qquad release\_mutex := release\_mutex \oplus \{(topWaiting!, mut?)\}, \\
\qquad\qquad release\_rcv := release\_rcv \oplus \{(topWaiting!, mut?)\}, \\
\qquad\qquad running\_task := topReady!, \\
\qquad\qquad state := state \oplus (\{(mutex\_holder(mut?), ready)\} \cup \\
\qquad\qquad\qquad (\{(topReady!, running)\} \cup \{(topWaiting!, ready)\})), \\
\qquad\qquad wait\_time := \{topWaiting!\} \lhd wait\_time, \\
\qquad\qquad wait\_rcv := \{topWaiting!\} \lhd wait\_rcv] \\
\qquad \wedge\ priority(topReady!) > priority(topWaiting!) \\
\qquad \wedge\ (st \in TASK \\
\qquad\qquad \wedge\ \neg\ (state \oplus (\{(mutex\_holder(mut?), ready)\} \cup \\
\qquad\qquad\qquad (\{(topReady!, running)\} \cup \\
\qquad\qquad\qquad\qquad \{(topWaiting!, ready)\}))) st = state(st) \\
\qquad\qquad\ \Rightarrow (state(st), (state \oplus \\
\qquad\qquad\qquad (\{(mutex\_holder(mut?), ready)\} \cup \\
\qquad\qquad\qquad\qquad (\{(topReady!, running)\} \cup \\
\qquad\qquad\qquad\qquad\qquad \{(topWaiting!, ready)\}))) st) \in transition) \\
\qquad \wedge\ (wr \in \mathrm{dom}\, wait\_rcv \wedge wait\_rcv(wr) = mut? \\
\qquad\qquad \Rightarrow priority(topWaiting!) \geq priority(wr)) \\
\qquad \Rightarrow rt \in TASK \\
\qquad\qquad \wedge\ state(rt) = ready \\
\qquad\qquad \wedge\ \neg\ priority(topReady!) \geq priority(rt))
\end{array}
$$

theorem MutexGiveWInhSR_TQTM_vc_ref

$\forall\, MutexGiveWInhSR\_TQTMFSBSig \mid true$

• pre $MutexGiveWInhSR\_TQTM$

MutexGiveWInhSW_TQTM

$\Delta\, TaskQueueTimeMutex$
$mut? : QUEUE$
$topWaiting! : TASK$

$running\_task \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv$
$mut? \in \operatorname{dom} mutex\_holder$
$running\_task = mutex\_holder(mut?)$
$mutex\_recursive(mut?) = 1$
$topWaiting! \in wait\_rcv^{\sim}(\!|\ \{mut?\}\ |\!)$
$\forall\, wr : wait\_rcv^{\sim}(\!|\ \{mut?\}\ |\!) \bullet priority(topWaiting!) \geq priority(wr)$
$\forall\, rt : state^{\sim}(\!|\ \{ready\}\ |\!) \bullet priority(topWaiting!) \geq priority(rt)$
$base\_priority(running\_task) \neq priority(running\_task)$
$priority(topWaiting!) > base\_priority(running\_task)$
$\exists\, st? : STATE;\ pri? : TASK \to \mathbb{N}$
$\quad | \ st? = ready \land pri? = priority \oplus$
$\quad\quad \{(running\_task \mapsto base\_priority(running\_task))\}$
$\quad \bullet\ Reschedule[topWaiting!/target?, tasks/tasks?]$
$queue' = queue$
$q\_max' = q\_max$
$q\_size' = q\_size \oplus \{(mut? \mapsto 1)\}$
$wait\_snd' = wait\_snd$
$wait\_rcv' = \{topWaiting!\} \lhd wait\_rcv$
$release\_snd' = release\_snd$
$release\_rcv' = release\_rcv \oplus \{(topWaiting! \mapsto mut?)\}$
$clock' = clock$
$delayed\_task' = delayed\_task$
$wait\_time' = \{topWaiting!\} \lhd wait\_time$
$time\_slice' = time\_slice$
$semaphore' = semaphore$
$mutex' = mutex$
$mutex\_holder' = \{mut?\} \lhd mutex\_holder$
$mutex\_recursive' = mutex\_recursive \oplus \{(mut? \mapsto 0)\}$
$basePriorityMan$
$release\_mutex' = release\_mutex \oplus \{(topWaiting! \mapsto mut?)\}$

MutexGiveWInhSW_TQTMFSBSig

$TaskQueueTimeMutex$
$mut? : QUEUE$

$running\_task \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv$
$mut? \in \operatorname{dom} mutex\_holder$
$running\_task = mutex\_holder(mut?)$
$mutex\_recursive(mut?) = 1$
$base\_priority(running\_task) \neq priority(running\_task)$
$\exists\, topWaiting! : wait\_rcv^{\sim}(\!|\ \{mut?\}\ |\!)$

$$\bullet\ (\forall\, wr : wait\_rcv^{\sim}(\!|\ \{mut?\}\ |\!)\ \bullet\ priority(topWaiting!) \geq priority(wr))$$
$$\wedge\ (\forall\, rt : state^{\sim}(\!|\ \{ready\}\ |\!)\ \bullet\ priority(topWaiting!) \geq priority(rt))$$
$$\wedge\ priority(topWaiting!) > base\_priority(running\_task)$$

---

theorem lMutexGiveWInhSW_TQTM_Lemma

$$\forall\, TaskQueueTimeMutex;\ topWaiting! : TASK;$$
$$mut? : QUEUE$$
$$\mid running\_task \notin \mathrm{dom}\ release\_snd \cup \mathrm{dom}\ release\_rcv$$
$$\wedge\ mut? \in \mathrm{dom}\ mutex\_holder$$
$$\wedge\ running\_task = mutex\_holder(mut?)$$
$$\wedge\ mutex\_recursive(mut?) = 1$$
$$\wedge\ topWaiting! \in wait\_rcv^{\sim}(\!|\ \{mut?\}\ |\!)$$
$$\wedge\ (\forall\, wrct : wait\_rcv^{\sim}(\!|\ \{mut?\}\ |\!)$$
$$\bullet\ priority(topWaiting!) \geq priority(wrct))$$
$$\wedge\ (\forall\, rtsk : state^{\sim}(\!|\ \{ready\}\ |\!)$$
$$\bullet\ priority(topWaiting!) \geq priority(rtsk))$$
$$\wedge\ base\_priority(running\_task) \neq priority(running\_task)$$
$$\wedge\ priority(topWaiting!) > base\_priority(running\_task)$$
$$\wedge\ running\_task \in \mathrm{ran}(\{mut?\} \lhd mutex\_holder)$$
$$\bullet\ \neg\ (TaskQueueTimeMutex[log\_context := log\_context \oplus$$
$$\{(mutex\_holder(wait\_rcv(topWaiting!)),$$
$$phys\_context)\},$$
$$mutex\_holder := \{wait\_rcv(topWaiting!)\} \lhd mutex\_holder,$$
$$mutex\_recursive := mutex\_recursive \oplus$$
$$\{(wait\_rcv(topWaiting!), 0)\},$$
$$phys\_context := log\_context(topWaiting!),$$
$$priority := priority \oplus$$
$$\{(mutex\_holder(wait\_rcv(topWaiting!)),$$
$$base\_priority(mutex\_holder($$
$$wait\_rcv(topWaiting!))))\},$$
$$q\_size := q\_size \oplus \{(wait\_rcv(topWaiting!), 1)\},$$
$$release\_mutex := release\_mutex \oplus$$
$$\{(topWaiting!, wait\_rcv(topWaiting!))\},$$
$$release\_rcv := release\_rcv \oplus$$
$$\{(topWaiting!, wait\_rcv(topWaiting!))\},$$
$$running\_task := topWaiting!,$$
$$state := state \oplus$$
$$(\{(mutex\_holder(wait\_rcv(topWaiting!)), ready)\}$$
$$\cup\{(topWaiting!, running)\}),$$
$$wait\_time := \{topWaiting!\} \lhd wait\_time,$$
$$wait\_rcv := \{topWaiting!\} \lhd wait\_rcv]$$
$$\wedge\ (st \in TASK$$
$$\wedge\ \neg\ (state \oplus$$
$$(\{(mutex\_holder(wait\_rcv(topWaiting!)), ready)\} \cup$$
$$\{(topWaiting!, running)\}))st = state(st)$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad \Rightarrow (state(st), (state \oplus \\
\qquad\qquad\qquad\qquad\qquad (\{(mutex\_holder(wait\_rcv(topWaiting!)), ready)\} \cup \\
\qquad\qquad\qquad\qquad\qquad\quad \{(topWaiting!, running)\}))st) \in transition) \\
\qquad\qquad\qquad\quad \wedge (wr \in \mathrm{dom}\ wait\_rcv \wedge wait\_rcv(wr) = wait\_rcv(topWaiting!) \\
\qquad\qquad\qquad\qquad \Rightarrow priority(topWaiting!) \geq priority(wr)) \\
\qquad\qquad\qquad\quad \Rightarrow rt \in TASK \\
\qquad\qquad\qquad\qquad \wedge\ state(rt) = ready \\
\qquad\qquad\qquad\qquad \wedge \neg\ priority(topWaiting!) \geq priority(rt))
\end{array}
$$

theorem lMutexGiveWInhSW_TQTM_Lemma1

$$
\begin{array}{l}
\forall\ TaskQueueTimeMutex;\ topWaiting! : TASK; \\
\qquad mut? : QUEUE \\
\qquad | \ running\_task \notin \mathrm{dom}\ release\_snd \cup \mathrm{dom}\ release\_rcv \\
\qquad \wedge\ mut? \in \mathrm{dom}\ mutex\_holder \\
\qquad \wedge\ running\_task = mutex\_holder(mut?) \\
\qquad \wedge\ mutex\_recursive(mut?) = 1 \\
\qquad \wedge\ topWaiting! \in wait\_rcv^{\sim}(\!|\ \{mut?\}\ |\!) \\
\qquad \wedge\ (\forall\ wrct : wait\_rcv^{\sim}(\!|\ \{mut?\}\ |\!) \\
\qquad\qquad \bullet\ priority(topWaiting!) \geq priority(wrct)) \\
\qquad \wedge\ (\forall\ rtsk : state^{\sim}(\!|\ \{ready\}\ |\!) \\
\qquad\qquad \bullet\ priority(topWaiting!) \geq priority(rtsk)) \\
\qquad \wedge\ base\_priority(running\_task) \neq priority(running\_task) \\
\qquad \wedge\ priority(topWaiting!) > base\_priority(running\_task) \\
\qquad \wedge\ running\_task \notin \mathrm{ran}(\{mut?\} \lhd mutex\_holder) \\
\qquad \bullet \neg\ (TaskQueueTimeMutex[\\
\qquad\qquad base\_priority := \{mutex\_holder(wait\_rcv(topWaiting!))\} \\
\qquad\qquad\qquad \lhd base\_priority, \\
\qquad\qquad log\_context := log\_context \oplus \\
\qquad\qquad\qquad \{(mutex\_holder(wait\_rcv(topWaiting!)), phys\_context)\}, \\
\qquad\qquad mutex\_holder := \{wait\_rcv(topWaiting!)\} \lhd mutex\_holder, \\
\qquad\qquad mutex\_recursive := mutex\_recursive \oplus \\
\qquad\qquad\qquad \{(wait\_rcv(topWaiting!), 0)\}, \\
\qquad\qquad phys\_context := log\_context(topWaiting!), \\
\qquad\qquad priority := priority \oplus \\
\qquad\qquad\qquad \{(mutex\_holder(wait\_rcv(topWaiting!)), \\
\qquad\qquad\qquad\qquad base\_priority(\\
\qquad\qquad\qquad\qquad\qquad mutex\_holder(wait\_rcv(topWaiting!))))\}, \\
\qquad\qquad q\_size := q\_size \oplus \{(wait\_rcv(topWaiting!), 1)\}, \\
\qquad\qquad release\_mutex := release\_mutex \oplus \\
\qquad\qquad\qquad \{(topWaiting!, wait\_rcv(topWaiting!))\}, \\
\qquad\qquad release\_rcv := release\_rcv \oplus \\
\qquad\qquad\qquad \{(topWaiting!, wait\_rcv(topWaiting!))\}, \\
\qquad\qquad running\_task := topWaiting!, \\
\qquad\qquad state := state \oplus \\
\qquad\qquad\qquad (\{(mutex\_holder(wait\_rcv(topWaiting!)), ready)\} \cup \\
\qquad\qquad\qquad\qquad \{(topWaiting!, running)\}),
\end{array}
$$

$$
\begin{array}{l}
\quad wait\_time := \{topWaiting!\} \lhd wait\_time, \\
\quad wait\_rcv := \{topWaiting!\} \lhd wait\_rcv] \\
\land\ (st \in TASK \\
\quad \land \neg\ (state\oplus \\
\qquad (\{(mutex\_holder(wait\_rcv(topWaiting!)), ready)\} \cup \\
\qquad \Rightarrow (state(st), (state\oplus \\
\qquad (\{(mutex\_holder(wait\_rcv(topWaiting!)), ready)\} \cup \\
\qquad \{(topWaiting!, running)\}))st) \in transition) \\
\land\ (wr \in \mathrm{dom}\ wait\_rcv \land wait\_rcv(wr) = wait\_rcv(topWaiting!) \\
\quad \Rightarrow priority(topWaiting!) \geq priority(wr)) \\
\Rightarrow rt \in TASK \\
\quad \land\ state(rt) = ready \\
\quad \land \neg\ priority(topWaiting!) \geq priority(rt))
\end{array}
$$

theorem MutexGiveWInhSW_TQTM_vc_ref

$\forall\ MutexGiveWInhSW\_TQTMFSBSig \mid true$

- pre $MutexGiveWInhSW\_TQTM$

MutexGiveNRecursive_TQTM

$\Delta TaskQueueTimeMutex$

$mut? : QUEUE$

$topReady! : TASK$

$running\_task \notin \mathrm{dom}\ release\_snd \cup \mathrm{dom}\ release\_rcv$

$mut? \in \mathrm{dom}\ mutex\_holder$

$running\_task = mutex\_holder(mut?)$

$mutex\_recursive(mut?) > 1$

$\Xi TaskQueueTime$

$semaphore' = semaphore$

$mutex' = mutex$

$mutex\_holder' = mutex\_holder$

$mutex\_recursive' = mutex\_recursive\oplus$
$\quad \{(mut? \mapsto mutex\_recursive(mut?) - 1)\}$

$\Xi OriginalPrioData$

$\Xi MReleasingData$

$topReady! = running\_task$

MutexGiveNRecursive_TQTMFSBSig

$TaskQueueTimeMutex$

$mut? : QUEUE$

$running\_task \notin \mathrm{dom}\ release\_snd \cup \mathrm{dom}\ release\_rcv$

$mut? \in \mathrm{dom}\ mutex\_holder$

$running\_task = mutex\_holder(mut?)$

$mutex\_recursive(mut?) > 1$

theorem MutexGiveNRecursive_TQTM_vc_ref
$\forall\ MutexGiveNRecursive\_TQTMFSBSig \mid true$
• pre $MutexGiveNRecursive\_TQTM$

$$
\begin{aligned}
MutexGive\_TQTM \mathrel{\widehat=}\ & MutexGiveNnonInh\_TQTM \\
\lor\ & MutexGiveNInhN\_TQTM \\
\lor\ & MutexGiveNInhS\_TQTM \\
\lor\ & MutexGiveWnonInhN\_TQTM \\
\lor\ & MutexGiveWnonInhS\_TQTM \\
\lor\ & MutexGiveWInhN\_TQTM \\
\lor\ & MutexGiveWInhSR\_TQTM \\
\lor\ & MutexGiveWInhSW\_TQTM \\
\lor\ & MutexGiveNRecursive\_TQTM
\end{aligned}
$$

Appendix H

SPECIFICATION FOR MULTI-CORE TASK MODEL

$$[CONTEXT, TASK, CORE]$$

$$\begin{array}{|l} bare\_context : CONTEXT \\ idles : \mathbb{F}\ TASK \\ cores : \mathbb{F}\ CORE \\ \hline \#cores = \#idle \\ cores \neq \emptyset \end{array}$$

$$STATE ::= nonexistent \mid ready \mid blocked \mid suspended \mid running$$

$$\begin{aligned} transition == & (\{blocked\} \times \{nonexistent, ready, running, suspended\}) \\ & \cup\ (\{nonexistent\} \times \{ready, running\}) \\ & \cup\ (\{ready\} \times \{nonexistent, running, suspended\}) \\ & \cup\ (\{running\} \times \{blocked, nonexistent, ready, suspended\}) \\ & \cup\ (\{suspended\} \times \{nonexistent, ready, running\}) \end{aligned}$$

$$\begin{array}{|l} slice\_delay : \mathbb{N} \\ \hline slice\_delay = 1 \end{array}$$

$$BOOL ::= TRUE \mid FALSE$$

TaskData

$$tasks : \mathbb{F}\ TASK$$
$$running\_tasks : cores \rightarrowtail TASK$$
$$executable : TASK \twoheadrightarrow cores$$

$$\operatorname{ran} running\_tasks \subseteq tasks$$
$$idles \subseteq tasks$$
$$\operatorname{dom} executable = tasks$$
$$\forall\, t : \operatorname{ran} running\_tasks \bullet running\_tasks^{\sim} t = executable(t)$$

Init_TaskData

$$TaskData'$$

$$tasks' = idles$$

StateData

$$state : TASK \rightarrow STATE$$

$$\forall\, i : idles \bullet state(i) \in \{ready, running\}$$

Init_StateData

$$StateData'$$

$$state' = (\lambda\, x : TASK \bullet nonexistent) \oplus (idles \times \{running\})$$

ContextData

$$phys\_context : cores \rightarrow CONTEXT$$
$$log\_context : TASK \rightarrow CONTEXT$$

Init_ContextData

$$ContextData'$$

$$phys\_context' = (\lambda\, c : cores \bullet bare\_context)$$
$$log\_context' = (\lambda\, x : TASK \bullet bare\_context)$$

PrioData

$$priority : TASK \rightarrow \mathbb{N}$$

$$\forall\, i : idles \bullet priority(i) = 0$$

Init_PrioData

$$PrioData'$$

$$priority' = (\lambda\, x : TASK \bullet 0)$$

Task

$$TaskData$$
$$StateData$$
$$ContextData$$
$$PrioData$$

$$tasks = TASK \setminus (state^{\sim}(\!|\, \{nonexistent\} \,|\!))$$
$$state^{\sim}(\!|\, \{running\} \,|\!) = \operatorname{ran} running\_tasks$$
$$\forall\, pt : state^{\sim}(\!|\, \{ready\} \,|\!);\ r : \operatorname{ran} running\_tasks \mid executable(pt) = executable(r) \bullet priority(r) \geq priority(pt)$$

Δ*Task*

$$Task$$
$$Task'$$

$$\forall\, st : TASK \mid state'(st) \neq state(st) \bullet state(st) \mapsto state'(st) \in transition$$

Init_Task

$$Task'$$

$$Init\_TaskData$$
$$Init\_StateData$$
$$Init\_ContextData$$
$$Init\_PrioData$$

createTaskSpeCoreN_T

$$\Delta Task$$
$$target? : TASK$$
$$newpri? : \mathbb{N}$$
$$executeCore : CORE$$

$$executeCore \in cores$$
$$state(target?) = nonexistent$$
$$newpri? \leq priority(running\_tasks(executeCore))$$
$$tasks' = tasks \cup \{target?\}$$
$$running\_tasks' = running\_tasks$$
$$executable' = executable \oplus \{(target? \mapsto executeCore)\}$$
$$state' = state \oplus \{(target? \mapsto ready)\}$$

$\Xi ContextData$
$priority' = priority \oplus \{(target? \mapsto newpri?)\}$

findACore_T

$\Delta Task$
$target? : TASK$
$newpri? : \mathbb{N}$
$executeCore? : CORE$
$executeCore : CORE$

$executeCore? \notin cores$
$executeCore \in cores$
$\exists\, tcs, cs : \mathbb{F}\, cores \mid$
$tcs = \{\, pc : cores \mid newpri? > priority(running\_tasks(pc))\,\}$
$\bullet\ (tcs = \emptyset \Rightarrow cs = cores) \land (tcs \neq \emptyset \Rightarrow cs = tcs)$
$\land\ (\forall\, oc : cs \bullet executeCore \in cs$
$\land\ \#(executable^{\sim}(\!|\, \{executeCore\}\, |\!)) \leq \#(executable^{\sim}(\!|\, \{oc\}\, |\!)))$

$CreateTaskN\_T \mathrel{\widehat=} ([executeCore?, executeCore : CORE \mid executeCore? \in cores \land executeCore = executeCore?] \lor findACore\_T) \land createTaskSpeCoreN\_T$

Reschedule

$\Delta Task$
$target? : TASK$
$tasks? : \mathbb{F}\, TASK$
$executable? : TASK \twoheadrightarrow cores$
$st? : STATE$
$pri? : TASK \to \mathbb{N}$

$tasks' = tasks?$
$running\_tasks' = running\_tasks \oplus \{(executable?(target?) \mapsto target?)\}$
$executable' = executable?$
$state' = state \oplus \{(target? \mapsto running),$
$(running\_tasks(executable?(target?)) \mapsto st?)\}$
$phys\_context' = phys\_context \oplus$
$\{(executable?(target?) \mapsto log\_context(target?))\}$
$log\_context' = log\_context \oplus$
$\{(running\_tasks(executable?(target?))$
$\mapsto phys\_context(executable?(target?)))\}$
$priority' = pri?$

$disableReschedule \mathrel{\widehat=} [Task \mid false] \land Reschedule$

createTaskSpeCoreS_T

$\Delta Task$
$target? : TASK$
$newpri? : \mathbb{N}$
$executeCore : CORE$

$executeCore \in cores$
$state(target?) = nonexistent$
$newpri? > priority(running\_tasks(executeCore))$
$\exists\, st? : STATE;\ tasks? : \mathbb{F}\, TASK;\ executable? : TASK \twoheadrightarrow cores;$
$\quad pri? : TASK \to \mathbb{N}$
$\quad \mid st? = ready \land tasks? = tasks \cup \{target?\}$
$\quad \land executable? = executable \oplus \{(target? \mapsto executeCore)\}$
$\quad \land pri? = priority \oplus \{(target? \mapsto newpri?)\} \bullet Reschedule$

$CreateTaskS\_T \mathrel{\widehat=} ([executeCore?, executeCore : CORE$
$\quad \mid executeCore? \in cores \land executeCore = executeCore?] \lor findACore\_T)$
$\quad \land createTaskSpeCoreS\_T$

$CreateTask\_T \mathrel{\widehat=} CreateTaskN\_T \lor CreateTaskS\_T$

DeleteTaskN_T

$\Delta Task$
$target? : TASK$
$topReady! : TASK$

$target? \in tasks \setminus idles$
$state(target?) \in \{ready, blocked, suspended\}$
$tasks' = tasks \setminus \{target?\}$
$running\_tasks' = running\_tasks$
$executable' = \{target?\} \lhd executable$
$state' = state \oplus \{(target? \mapsto nonexistent)\}$
$phys\_context' = phys\_context$
$log\_context' = log\_context \oplus \{(target? \mapsto bare\_context)\}$
$\Xi PrioData$
$topReady! = running\_tasks(executable(target?))$

findTopReady

$Task$
$target? : TASK$
$topReady! : TASK$

$state(topReady!) = ready$
$executable(topReady!) = executable(target?)$

$$\forall\, rt : state^{\sim}(\!|\, \{ready\} \,|\!) \mid executable(rt) = executable(topReady!) \bullet priority(topReady!) \geq priority(rt)$$

DeleteTaskS_T

$\Delta Task$
$target? : TASK$
$topReady! : TASK$

$target? \in tasks \setminus idles$
$state(target?) = running$
$findTopReady$
$tasks' = tasks \setminus \{target?\}$
$running\_tasks' = running\_tasks \oplus \{executable(target?) \mapsto topReady!\}$
$executable' = \{target?\} \lhd executable$
$state' = state \oplus \{(topReady! \mapsto running), (target? \mapsto nonexistent)\}$
$phys\_context' = phys\_context \oplus \{(executable(target?) \mapsto log\_context(topReady!))\}$
$log\_context' = log\_context \oplus \{(target? \mapsto bare\_context)\}$
$\Xi PrioData$

$DeleteTask\_T \mathrel{\widehat=} DeleteTaskN\_T \lor DeleteTaskS\_T$

ExecuteRunningTask_T

$\Delta Task$
$target! : \mathbb{F}\ TASK$

$\Xi TaskData$
$\Xi StateData$
$\forall\, c : cores \bullet phys\_context'(c) \neq phys\_context(c)$
$log\_context' = log\_context$
$\Xi PrioData$
$target! = \operatorname{ran} running\_tasks$

SuspendTaskN_T

$\Delta Task$
$target? : TASK$
$topReady! : TASK$

$state(target?) \in \{ready, blocked\}$
$target? \notin idles$
$\Xi TaskData$
$state' = state \oplus \{(target? \mapsto suspended)\}$
$\Xi ContextData$

$\Xi PrioData$
$topReady! = running\_tasks(executable(target?))$

SuspendTaskS_T

$\Delta Task$
$target? : TASK$
$topReady! : TASK$

$state(target?) = running$
$target? \notin idles$
$findTopReady$
$\exists\, st? : STATE \mid st? = suspended$
$\bullet\ Reschedule[tasks/tasks?, executable/executable?, priority/pri?,$
$topReady!/target?]$

SuspendTaskO_T

$\Xi Task$
$target? : TASK$
$topReady! : TASK$

$state(target?) = suspended$
$topReady! = running\_tasks(executable(target?))$

$SuspendTask\_T \mathrel{\widehat=} SuspendTaskN\_T$
$\lor\ SuspendTaskS\_T$
$\lor\ SuspendTaskO\_T$

ResumeTaskN_T

$\Delta Task$
$target? : TASK$

$state(target?) = suspended$
$priority(target?) \leq priority(running\_tasks(executable(target?)))$
$\Xi TaskData$
$state' = state \oplus \{(target? \mapsto ready)\}$
$\Xi ContextData$
$\Xi PrioData$

ResumeTaskS_T

$\Delta Task$
$target? : TASK$

$state(target?) = suspended$
$priority(target?) > priority(running\_tasks(executable(target?)))$
$\exists\, st? : STATE \mid st? = ready$
$\quad \bullet\ Reschedule[tasks/tasks?, executable/executable?, priority/pri?]$

$ResumeTask\_T \mathrel{\widehat=} ResumeTaskN\_T \lor ResumeTaskS\_T$

ChangeTaskPriorityN_T

$\Delta Task$
$newpri? : \mathbb{N}$
$target? : TASK$
$topReady! : TASK$

$state(target?) = ready$
$\quad \Rightarrow newpri? \leq priority(running\_tasks(executable(target?)))$
$state(target?) = running \Rightarrow$
$\quad (\forall\, rt : state^{\sim}(\!|\ \{ready\}\ |\!) \mid executable(rt) = executable(target?)$
$\quad \bullet\ newpri? \geq priority(rt))$
$state(target?) \neq nonexistent$
$target? \in idles \Rightarrow newpri? = 0$
$\Xi TaskData$
$\Xi StateData$
$\Xi ContextData$
$priority' = priority \oplus \{(target? \mapsto newpri?)\}$
$topReady! = running\_tasks(executable(target?))$

ChangeTaskPriorityS_T

$\Delta Task$
$newpri? : \mathbb{N}$
$target? : TASK$
$topReady! : TASK$

$state(target?) = ready$
$newpri? > priority(running\_tasks(executable(target?)))$
$target? \in idles \Rightarrow newpri? = 0$
$\exists\, st? : STATE;\ pri? : TASK \to \mathbb{N}$
$\quad \mid st? = ready \land pri? = priority \oplus \{(target? \mapsto newpri?)\}$
$\quad \bullet\ Reschedule[tasks/tasks?, executable/executable?]$
$topReady! = target?$

ChangeTaskPriorityD_T

$\Delta Task$
$newpri? : \mathbb{N}$

$$
\begin{array}{l}
target? : TASK \\
topReady! : TASK \\
\hline
state(target?) = running \\
target? \in idles \Rightarrow newpri? = 0 \\
findTopReady \\
newpri? < priority(topReady!) \\
\exists\, st? : STATE;\ pri? : TASK \to \mathbb{N} \\
\quad \mid st? = ready \land pri? = priority \oplus \{(target? \mapsto newpri?)\} \\
\quad \bullet\ Reschedule[tasks/tasks?, executable/executable?, topReady!/target?]
\end{array}
$$

$$
\begin{array}{lll}
ChangeTaskPriority\_T & \widehat{=} & ChangeTaskPriorityN\_T \\
 & \lor & ChangeTaskPriorityS\_T \\
 & \lor & ChangeTaskPriorityD\_T
\end{array}
$$

$$
\begin{array}{l}
MigrationN\_T \\
\hline
\Delta Task \\
target? : TASK \\
topReady! : TASK \\
newCore? : cores \\
\hline
state(target?) \in \{ready, blocked, suspended\} \\
state(target?) = ready \Rightarrow \\
\quad priority(target?) \leq priority(running\_tasks(newCore?)) \\
target? \notin idles \\
newCore? \neq executable(target?) \\
tasks' = tasks \\
running\_tasks' = running\_tasks \\
executable' = executable \oplus \{(target? \mapsto newCore?)\} \\
\Xi StateData \\
\Xi ContextData \\
\Xi PrioData \\
topReady! = running\_tasks(newCore?)
\end{array}
$$

$$
\begin{array}{l}
MigrationS\_T \\
\hline
\Delta Task \\
target? : TASK \\
topReady! : TASK \\
newCore? : cores \\
\hline
state(target?) = ready \\
priority(target?) > priority(running\_tasks(newCore?)) \\
target? \notin idles \\
newCore? \neq executable(target?)
\end{array}
$$

$$\exists\, st? : STATE;\ executable? : TASK \twoheadrightarrow cores \mid st? = ready \land executable? = executable \oplus \{(target? \mapsto newCore?)\} \bullet Reschedule[tasks/tasks?, priority/pri?]$$

$$topReady! = target?$$

MigrationRuN_T

$$\Delta Task$$

$$target? : TASK$$

$$topReady! : TASK$$

$$newCore? : cores$$

$$state(target?) = running$$

$$priority(target?) \leq priority(running\_tasks(newCore?))$$

$$target? \notin idles$$

$$newCore? \neq executable(target?)$$

$$findTopReady$$

$$\exists\, st? : STATE;\ executable? : TASK \twoheadrightarrow cores \mid st? = ready \land executable? = executable \oplus \{(target? \mapsto newCore?)\} \bullet Reschedule[tasks/tasks?, priority/pri?]$$

MigrationRuS_T

$$\Delta Task$$

$$target? : TASK$$

$$topReady! : TASK$$

$$newCore? : cores$$

$$state(target?) = running$$

$$priority(target?) > priority(running\_tasks(newCore?))$$

$$target? \notin idles$$

$$newCore? \neq executable(target?)$$

$$findTopReady$$

$$tasks' = tasks$$

$$running\_tasks' = running\_tasks \oplus \{(executable(target?) \mapsto topReady!), (newCore? \mapsto target?)\}$$

$$executable' = executable \oplus \{(target? \mapsto newCore?)\}$$

$$state' = state \oplus \{(topReady! \mapsto running), (running\_tasks(newCore?) \mapsto ready)\}$$

$$phys\_context' = phys\_context \oplus \{(executable(topReady!) \mapsto log\_context(topReady!)), (newCore? \mapsto phys\_context(executable(topReady!)))\}$$

$$log\_context' = log\_context \oplus \{(running\_tasks(newCore?) \mapsto phys\_context(newCore?))\}$$

$$priority' = priority$$

$$
\begin{aligned}
Migration\_T \triangleq\ & MigrationN\_T \\
\lor\ & MigrationS\_T \\
\lor\ & MigrationRuN\_T \\
\lor\ & MigrationRuS\_T
\end{aligned}
$$

Appendix I

SPECIFICATION FOR MULTI-CORE QUEUE MODEL

$[QUEUE]$

QueueData

$$
\begin{array}{l}
queue : \mathbb{P}\, QUEUE \\
q\_max : QUEUE \pfun \mathbb{N}_1 \\
q\_size : QUEUE \pfun \mathbb{N} \\
q\_ava : QUEUE \pfun \mathbb{F}\, CORE \\
\hline
\mathrm{dom}\, q\_max = \mathrm{dom}\, q\_size \\
\mathrm{dom}\, q\_size = \mathrm{dom}\, q\_ava \\
\mathrm{dom}\, q\_ava = queue \\
\mathrm{ran}\, q\_ava \subseteq \mathbb{F}\, cores \\
\forall\, q : QUEUE \mid q \in queue \bullet q\_size(q) \leq q\_max(q)
\end{array}
$$

Init_QueueData

$$
\begin{array}{l}
QueueData' \\
\hline
queue' = \emptyset \\
q\_max' = \emptyset \\
q\_size' = \emptyset \\
q\_ava' = \emptyset
\end{array}
$$

WaitingData

$$
wait\_snd : TASK \pfun QUEUE
$$

$$wait\_rcv : TASK \pfun QUEUE$$

$$\dom wait\_snd \cap \dom wait\_rcv = \emptyset$$

Init_WaitingData

$$WaitingData'$$

$$wait\_snd' = \emptyset$$
$$wait\_rcv' = \emptyset$$

QReleasingData

$$release\_snd : TASK \pfun QUEUE$$
$$release\_rcv : TASK \pfun QUEUE$$

$$\dom release\_snd \cap \dom release\_rcv = \emptyset$$

Init_QReleasingData

$$QReleasingData'$$

$$release\_snd' = \emptyset$$
$$release\_rcv' = \emptyset$$

Queue

$$QueueData$$
$$WaitingData$$
$$QReleasingData$$

$$\ran wait\_snd \subseteq queue$$
$$\ran wait\_rcv \subseteq queue$$
$$\ran release\_snd \subseteq queue$$
$$\ran release\_rcv \subseteq queue$$
$$(\dom wait\_snd \cup \dom wait\_rcv)$$
$$\quad \cap (\dom release\_snd \cap \dom release\_rcv) = \emptyset$$

Init_Queue

$$Queue'$$

$$Init\_QueueData$$
$$Init\_WaitingData$$
$$Init\_QReleasingData$$

TaskQueue

$Task$
$Queue$

$\mathrm{dom}\ wait\_snd \subseteq state^{\sim}(\!|\ \{blocked\}\ |\!)$
$\mathrm{dom}\ wait\_rcv \subseteq state^{\sim}(\!|\ \{blocked\}\ |\!)$

Init_TaskQueue

$TaskQueue'$

$Init\_Task$
$Init\_Queue$

$\Delta TaskQueue \mathrel{\widehat=} TaskQueue \land TaskQueue' \land \Delta Task$

ExtendTaskXi

$\Delta TaskQueue$
$self? : TASK$

$state(self?) = running$
$self? \notin \mathrm{dom}\ release\_snd \cup \mathrm{dom}\ release\_rcv$
$\Xi Queue$

$CreateTask\_TQ \mathrel{\widehat=} ExtendTaskXi \land CreateTask\_T$

DeleteTaskN_TQ

$DeleteTaskN\_T$
$\Delta TaskQueue$
$self? : TASK$

$self? \notin \mathrm{dom}\ release\_snd \cup \mathrm{dom}\ release\_rcv$
$state(self?) = running$
$\Xi QueueData$
$wait\_snd' = \{target?\} \lhd wait\_snd$
$wait\_rcv' = \{target?\} \lhd wait\_rcv$
$release\_snd' = \{target?\} \lhd release\_snd$
$release\_rcv' = \{target?\} \lhd release\_rcv$

$DeleteTaskS\_TQ \mathrel{\widehat=} ExtendTaskXi \land DeleteTaskS\_T$

$DeleteTask\_TQ \mathrel{\widehat=} DeleteTaskN\_TQ \lor DeleteTaskS\_TQ$

$$ExecuteRunningTask\_TQ \mathrel{\widehat=} ExtendTaskXi \land ExecuteRunningTask\_T$$

SuspendTaskN_TQ

$SuspendTaskN\_T$
$\Delta TaskQueue$
$self? : TASK$

$self? \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv$
$state(self?) = running$
$\Xi QueueData$
$wait\_snd' = \{target?\} \lhd wait\_snd$
$wait\_rcv' = \{target?\} \lhd wait\_rcv$
$\Xi QReleasingData$

$$SuspendTask\_TQ \mathrel{\widehat=} SuspendTaskN\_TQ \\ \lor (ExtendTaskXi \land (SuspendTaskS\_T \lor SuspendTaskO\_T))$$

$$ResumeTask\_TQ \mathrel{\widehat=} ExtendTaskXi \land ResumeTask\_T$$

$$ChangeTaskPriority\_TQ \mathrel{\widehat=} ExtendTaskXi \land ChangeTaskPriority\_T$$

$$Migration\_TQ \mathrel{\widehat=} ExtendTaskXi \land Migration\_T$$

CreateQueue_TQ

$\Delta TaskQueue$
$que? : QUEUE$
$self? : TASK$
$cset? : \mathbb{F}\, cores$
$size? : \mathbb{N}$

$self? \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv$
$state(self?) = running$
$que? \notin queue$
$size? > 0$
$cset? \neq \emptyset$
$\Xi Task$
$queue' = queue \cup \{que?\}$
$q\_max' = q\_max \oplus \{(que? \mapsto size?)\}$
$q\_size' = q\_size \oplus \{(que? \mapsto 0)\}$
$q\_ava' = q\_ava \oplus \{(que? \mapsto cset?)\}$
$\Xi WaitingData$
$\Xi QReleasingData$

DeleteQueue_TQ

$\Delta TaskQueue$
$que? : QUEUE$
$self? : TASK$

$self? \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv$
$state(self?) = running$
$que? \in queue$
$executable(self?) \in q\_ava(que?)$
$que? \notin \operatorname{ran} wait\_snd \cup \operatorname{ran} wait\_rcv$
$que? \notin \operatorname{ran} release\_snd \cup \operatorname{ran} release\_rcv$
$\Xi Task$
$queue' = queue \setminus \{que?\}$
$q\_max' = \{que?\} \lhd q\_max$
$q\_size' = \{que?\} \lhd q\_size$
$q\_ava' = \{que?\} \lhd q\_ava$
$\Xi WaitingData$
$\Xi QReleasingData$

QueueSendN_TQ

$\Delta TaskQueue$
$que? : QUEUE$
$self? : TASK$
$topReady! : TASK$

$self? \notin \operatorname{dom} release\_rcv$
$self? \in \operatorname{dom} release\_snd \Rightarrow que? = release\_snd(self?)$
$que? \in queue$
$q\_size(que?) < q\_max(que?)$
$state(self?) = running$
$executable(self?) \in q\_ava(que?)$
$que? \notin \operatorname{ran} wait\_rcv$
$\Xi Task$
$queue' = queue$
$q\_max' = q\_max$
$q\_size' = q\_size \oplus \{(que? \mapsto q\_size(que?) + 1)\}$
$q\_ava' = q\_ava$
$\Xi WaitingData$
$release\_snd' = \{self?\} \lhd release\_snd$
$release\_rcv' = release\_rcv$
$topReady! = self?$

QueueSendF_TQ

$\Delta TaskQueue$
$que? : QUEUE$

$self? : TASK$
$topReady! : TASK$

$self? \notin \operatorname{dom} release\_rcv$
$self? \in \operatorname{dom} release\_snd \Rightarrow que? = release\_snd(self?)$
$que? \in queue$
$q\_size(que?) = q\_max(que?)$
$state(self?) = running$
$executable(self?) \in q\_ava(que?)$
$self? \notin idles$
$findTopReady[self?/target?]$
$\exists\, st? : STATE \mid st? = blocked$
$\quad \bullet\ Reschedule[topReady!/target?, tasks/tasks?,$
$\qquad executable/executable?, priority/pri?]$
$\Xi QueueData$
$wait\_snd' = wait\_snd \oplus \{(self? \mapsto que?)\}$
$wait\_rcv' = wait\_rcv$
$release\_snd' = \{self?\} \lhd release\_snd$
$release\_rcv' = release\_rcv$

QueueSendW_TQ

$\Delta TaskQueue$
$que? : QUEUE$
$self? : TASK$
$topReady! : TASK$

$self? \notin \operatorname{dom} release\_rcv$
$self? \in \operatorname{dom} release\_snd \Rightarrow que? = release\_snd(self?)$
$que? \in queue$
$state(self?) = running$
$executable(self?) \in q\_ava(que?)$
$topReady! \in wait\_rcv^{\sim}(\!|\ \{que?\}\ |\!)$
$\forall\, wr : wait\_rcv^{\sim}(\!|\ \{que?\}\ |\!) \bullet priority(topReady!) \geq priority(wr)$
$priority(topReady!) \leq priority(running\_tasks(executable(topReady!)))$
$\Xi TaskData$
$state' = state \oplus \{(topReady! \mapsto ready)\}$
$\Xi ContextData$
$\Xi PrioData$
$queue' = queue$
$q\_max' = q\_max$
$q\_size' = q\_size \oplus \{(que? \mapsto 1)\}$
$q\_ava' = q\_ava$
$wait\_snd' = wait\_snd$
$wait\_rcv' = \{topReady!\} \lhd wait\_rcv$
$release\_snd' = \{self?\} \lhd release\_snd$
$release\_rcv' = release\_rcv \oplus \{(topReady! \mapsto que?)\}$

QueueSendWS_TQ

$\Delta TaskQueue$
$que? : QUEUE$
$self? : TASK$
$topReady! : TASK$

$self? \notin \operatorname{dom} release\_rcv$
$self? \in \operatorname{dom} release\_snd \Rightarrow que? = release\_snd(self?)$
$que? \in queue$
$state(self?) = running$
$executable(self?) \in q\_ava(que?)$
$self? \notin idles$
$topReady! \in wait\_rcv^{\sim}(\!| \{que?\} |\!)$
$\forall wr : wait\_rcv^{\sim}(\!| \{que?\} |\!) \bullet priority(topReady!) \geq priority(wr)$
$priority(topReady!) > priority(running\_tasks(executable(topReady!)))$
$\exists st? : STATE \mid st? = ready$
$\quad \bullet Reschedule[topReady!/target?, tasks/tasks?,$
$\quad\quad executable/executable?, priority/pri?]$
$queue' = queue$
$q\_max' = q\_max$
$q\_size' = q\_size \oplus \{(que? \mapsto 1)\}$
$q\_ava' = q\_ava$
$wait\_snd' = wait\_snd$
$wait\_rcv' = \{topReady!\} \lhd wait\_rcv$
$release\_snd' = \{self?\} \lhd release\_snd$
$release\_rcv' = release\_rcv \oplus \{(topReady! \mapsto que?)\}$

$$QueueSend\_TQ \mathrel{\widehat=} QueueSendN\_TQ \lor QueueSendF\_TQ \lor QueueSendW\_TQ \lor QueueSendWS\_TQ$$

QueueReceiveN_TQ

$\Delta TaskQueue$
$que? : QUEUE$
$self? : TASK$
$topReady! : TASK$

$self? \notin \operatorname{dom} release\_snd$
$self? \in \operatorname{dom} release\_rcv \Rightarrow que? = release\_rcv(self?)$
$que? \in queue$
$q\_size(que?) \neq 0$
$state(self?) = running$

$executable(self?) \in q\_ava(que?)$
$que? \notin \operatorname{ran} wait\_snd$
$\Xi Task$
$queue' = queue$
$q\_max' = q\_max$
$q\_size' = q\_size \oplus \{(que? \mapsto q\_size(que?) - 1)\}$
$q\_ava' = q\_ava$
$\Xi WaitingData$
$release\_snd' = release\_snd$
$release\_rcv' = \{self?\} \lhd release\_rcv$
$topReady! = self?$

QueueReceiveE_TQ

$\Delta TaskQueue$
$que? : QUEUE$
$self? : TASK$
$topReady! : TASK$

$self? \notin \operatorname{dom} release\_snd$
$self? \in \operatorname{dom} release\_rcv \Rightarrow que? = release\_rcv(self?)$
$que? \in queue$
$q\_size(que?) = 0$
$state(self?) = running$
$executable(self?) \in q\_ava(que?)$
$self? \notin idles$
$findTopReady[self?/target?]$
$\exists st? : STATE \mid st? = blocked$
• $Reschedule[topReady!/target?, tasks/tasks?,$
$executable/executable?, priority/pri?]$
$\Xi QueueData$
$wait\_snd' = wait\_snd$
$wait\_rcv' = wait\_rcv \oplus \{(self? \mapsto que?)\}$
$release\_snd' = release\_snd$
$release\_rcv' = \{self?\} \lhd release\_rcv$

QueueReceiveW_TQ

$\Delta TaskQueue$
$que? : QUEUE$
$self? : TASK$
$topReady! : TASK$

$self? \notin \operatorname{dom} release\_snd$
$self? \in \operatorname{dom} release\_rcv \Rightarrow que? = release\_rcv(self?)$
$que? \in queue$
$state(self?) = running$

$$
\begin{array}{l}
executable(self?) \in q\_ava(que?) \\
topReady! \in wait\_snd^{\sim}(\!| \{que?\} |\!) \\
\forall ws : wait\_snd^{\sim}(\!| \{que?\} |\!) \bullet priority(topReady!) \geq priority(ws) \\
priority(topReady!) \leq priority(running\_tasks(executable(topReady!))) \\
\Xi TaskData \\
state' = state \oplus \{(topReady! \mapsto ready)\} \\
\Xi ContextData \\
\Xi PrioData \\
queue' = queue \\
q\_max' = q\_max \\
q\_size' = q\_size \oplus \{(que? \mapsto q\_max(que?) - 1)\} \\
q\_ava' = q\_ava \\
wait\_snd' = \{topReady!\} \lhd wait\_snd \\
wait\_rcv' = wait\_rcv \\
release\_snd' = release\_snd \oplus \{(self? \mapsto que?)\} \\
release\_rcv' = \{self?\} \lhd release\_rcv
\end{array}
$$

QueueReceiveWS_TQ

$$
\begin{array}{l}
\Delta TaskQueue \\
que? : QUEUE \\
self? : TASK \\
topReady! : TASK \\
\hline
self? \notin \operatorname{dom} release\_snd \\
self? \in \operatorname{dom} release\_rcv \Rightarrow que? = release\_rcv(self?) \\
que? \in queue \\
state(self?) = running \\
executable(self?) \in q\_ava(que?) \\
self? \notin idles \\
topReady! \in wait\_snd^{\sim}(\!| \{que?\} |\!) \\
\forall ws : wait\_snd^{\sim}(\!| \{que?\} |\!) \bullet priority(topReady!) \geq priority(ws) \\
priority(topReady!) > priority(running\_tasks(executable(topReady!))) \\
\exists st? : STATE \mid st? = ready \\
\quad \bullet\ Reschedule[topReady!/target?, tasks/tasks?, \\
\qquad executable/executable?, priority/pri?] \\
queue' = queue \\
q\_max' = q\_max \\
q\_size' = q\_size \oplus \{(que? \mapsto q\_max(que?) - 1)\} \\
q\_ava' = q\_ava \\
wait\_snd' = \{topReady!\} \lhd wait\_snd \\
wait\_rcv' = wait\_rcv \\
release\_snd' = release\_snd \oplus \{(topReady! \mapsto que?)\} \\
release\_rcv' = \{self?\} \lhd release\_rcv
\end{array}
$$

$$
\begin{aligned}
QueueReceive\_TQ \mathrel{\widehat=}\ & QueueReceiveN\_TQ \\
\lor\ & QueueReceiveE\_TQ \\
\lor\ & QueueReceiveW\_TQ \\
\lor\ & QueueReceiveWS\_TQ
\end{aligned}
$$

ChangeQueueLevel_TQ

$\Delta TaskQueue$
$que? : QUEUE$
$self? : TASK$
$cset? : \mathbb{F}\ cores$

$self? \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv$
$que? \in queue$
$state(self?) = running$
$executable(self?) \in q\_ava(que?)$
$cset? \neq q\_ava(que?)$
$cset? \neq \emptyset$
$\forall\, t : wait\_rcv^{\sim}(\!|\ \{que?\}\ |\!) \cup wait\_snd^{\sim}(\!|\ \{que?\}\ |\!)$
$\quad \bullet\ executable(t) \in cset?$
$\Xi Task$
$queue' = queue$
$q\_max' = q\_max$
$q\_size' = q\_size$
$q\_ava' = q\_ava \oplus \{(que? \mapsto cset?)\}$
$\Xi WaitingData$
$\Xi QReleasingData$

Appendix J

SPECIFICATION FOR MULTI-CORE TIME MODEL

Time

$$clock : \mathbb{N}$$
$$delayed\_task : \mathbb{P}\ TASK$$
$$time : TASK \pfun \mathbb{N}$$
$$time\_slice : \mathbb{N}$$

$$\forall\, t : \mathrm{dom}\ time \bullet time(t) \geq clock$$

Init_Time

$$Time'$$

$$clock' = 0$$
$$delayed\_task' = \emptyset$$
$$time' = \emptyset$$
$$time\_slice' = slice\_delay$$

TaskQueueTime

$$TaskQueue$$
$$Time$$

$$\langle delayed\_task, \mathrm{dom}\ wait\_snd, \mathrm{dom}\ wait\_rcv \rangle \ \mathsf{partition}\ \mathrm{dom}\ time$$
$$delayed\_task \subseteq state^{\sim}(\!|\ \{blocked\}\ |\!)$$

Init_TaskQueueTime

$$TaskQueueTime'$$

$$Init\_TaskQueue$$
$$Init\_Time$$

$$\Delta TaskQueueTime \mathrel{\widehat=} TaskQueueTime \land TaskQueueTime' \land \Delta Task$$

ExtendTaskQueueXi

$$\Delta TaskQueueTime$$

$$\Xi Time$$

$$CreateTask\_TQT \mathrel{\widehat=} ExtendTaskQueueXi \land CreateTask\_TQ$$

DeleteTaskN_TQT

$$DeleteTaskN\_TQ$$
$$\Delta TaskQueueTime$$

$$clock' = clock$$
$$delayed\_task' = delayed\_task \setminus \{target?\}$$
$$time' = \{target?\} \lhd time$$
$$time\_slice' = time\_slice$$

$$DeleteTaskS\_TQT \mathrel{\widehat=} ExtendTaskQueueXi \land DeleteTaskS\_TQ$$

$$DeleteTask\_TQT \mathrel{\widehat=} DeleteTaskN\_TQT \lor DeleteTaskS\_TQT$$

$$ExecuteRunningTask\_TQT \mathrel{\widehat=} ExtendTaskQueueXi \land ExecuteRunningTask\_TQ$$

SuspendTaskN_TQT

$$SuspendTaskN\_TQ$$
$$\Delta TaskQueueTime$$

$$clock' = clock$$
$$delayed\_task' = delayed\_task \setminus \{target?\}$$
$$time' = \{target?\} \lhd time$$
$$time\_slice' = time\_slice$$

$$SuspendTask\_TQT \mathrel{\widehat=} SuspendTaskN\_TQT \lor (ExtendTaskQueueXi \land (SuspendTaskS\_T \lor SuspendTaskO\_T))$$

$$ResumeTask\_TQT \mathrel{\widehat=} ExtendTaskQueueXi \land ResumeTask\_TQ$$

$$ChangeTaskPriority\_TQT \mathrel{\widehat=} ExtendTaskQueueXi \land ChangeTaskPriority\_TQ$$

$$Migration\_TQT \mathrel{\widehat=} ExtendTaskQueueXi \land Migration\_TQ$$

$$CreateQueue\_TQT \mathrel{\widehat=} ExtendTaskQueueXi \land CreateQueue\_TQ$$

$$DeleteQueue\_TQT \mathrel{\widehat=} ExtendTaskQueueXi \land DeleteQueue\_TQ$$

$$QueueSendN\_TQT \mathrel{\widehat=} ExtendTaskQueueXi \land QueueSendN\_TQ$$

QueueSendF_TQT

$$QueueSendF\_TQ$$
$$\Delta TaskQueueTime$$
$$n? : \mathbb{N}$$

$$n? > clock$$
$$clock' = clock$$
$$delayed\_task' = delayed\_task$$
$$time' = time \oplus \{(self? \mapsto n?)\}$$
$$time\_slice' = time\_slice$$

QueueSendW_TQT

$$QueueSendW\_TQ$$
$$\Delta TaskQueueTime$$

$$clock' = clock$$
$$delayed\_task' = delayed\_task$$
$$time' = \{topReady!\} \lhd time$$
$$time\_slice' = time\_slice$$

QueueSendWS_TQT

$$QueueSendWS\_TQ$$
$$\Delta TaskQueueTime$$

$$clock' = clock$$

$$
\begin{array}{l}
delayed\_task' = delayed\_task \\
time' = \{topReady!\} \lhd time \\
time\_slice' = time\_slice
\end{array}
$$

$$
\begin{array}{l}
QueueSend\_TQT \mathrel{\widehat=} QueueSendN\_TQT \\
\qquad\qquad \lor\ QueueSendF\_TQT \\
\qquad\qquad \lor\ QueueSendW\_TQT \\
\qquad\qquad \lor\ QueueSendWS\_TQT
\end{array}
$$

$$
QueueReceiveN\_TQT \mathrel{\widehat=} ExtendTaskQueueXi \land QueueReceiveN\_TQ
$$

QueueReceiveE_TQT

$$
\begin{array}{l}
QueueReceiveE\_TQ \\
\Delta TaskQueueTime \\
n? : \mathbb{N} \\
\hline
n? > clock \\
clock' = clock \\
delayed\_task' = delayed\_task \\
time' = time \oplus \{(self? \mapsto n?)\} \\
time\_slice' = time\_slice
\end{array}
$$

QueueReceiveW_TQT

$$
\begin{array}{l}
QueueReceiveW\_TQ \\
\Delta TaskQueueTime \\
\hline
clock' = clock \\
delayed\_task' = delayed\_task \\
time' = \{topReady!\} \lhd time \\
time\_slice' = time\_slice
\end{array}
$$

QueueReceiveWS_TQT

$$
\begin{array}{l}
QueueReceiveWS\_TQ \\
\Delta TaskQueueTime \\
\hline
clock' = clock \\
delayed\_task' = delayed\_task \\
time' = \{topReady!\} \lhd time \\
time\_slice' = time\_slice
\end{array}
$$

$$
\begin{array}{l}
QueueReceive\_TQT \mathrel{\widehat=} QueueReceiveN\_TQT \lor QueueReceiveE\_TQT \\
\qquad\qquad \lor\ QueueReceiveW\_TQT \lor QueueReceiveWS\_TQT
\end{array}
$$

$$ChangeQueueLevel\_TQT \mathrel{\widehat=} ExtendTaskQueueXi \land ChangeQueueLevel\_TQ$$

DelayUntil_TQT

$\Delta TaskQueueTime$
$n? : \mathbb{N}$
$self? : TASK$
$topReady! : TASK$

$self? \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv$
$state(self?) = running$
$self? \notin idles$
$findTopReady[self?/target?]$
$n? > clock$
$\exists\, st? : STATE \mid st? = blocked$
$\quad \bullet\ Reschedule[topReady!/target?, tasks/tasks?,$
$\qquad executable/executable?, priority/pri?]$
$\Xi Queue$
$clock' = clock$
$delayed\_task' = delayed\_task \cup \{self?\}$
$time' = time \oplus \{(self? \mapsto n?)\}$
$time\_slice' = time\_slice$

CheckDelayedTaskN_TQT

$\Delta TaskQueueTime$
$topWaiting! : TASK$
$self? : TASK$

$self? \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv$
$state(self?) = running$
$topWaiting! \in \operatorname{dom} time$
$\forall\, wt : \operatorname{dom} time \bullet time(topWaiting!) \leq time(wt)$
$\forall\, wt : \operatorname{dom} time \mid time(wt) = time(topWaiting!)$
$\quad \bullet\ priority(topWaiting!) \geq priority(wt)$
$priority(topWaiting!)$
$\quad \leq priority(running\_tasks(executable(topWaiting!)))$
$\Xi TaskData$
$state' = state \oplus \{(topWaiting! \mapsto ready)\}$
$\Xi ContextData$
$\Xi PrioData$
$\Xi QueueData$
$wait\_snd' = \{topWaiting!\} \lhd wait\_snd$
$wait\_rcv' = \{topWaiting!\} \lhd wait\_rcv$
$\Xi QReleasingData$
$clock' = time(topWaiting!)$
$delayed\_task' = delayed\_task \setminus \{topWaiting!\}$

$time' = \{topWaiting!\} \lhd time$
$time\_slice' = time\_slice$

CheckDelayedTaskS_TQT

$\Delta TaskQueueTime$
$topWaiting! : TASK$
$self? : TASK$

$self? \notin \mathrm{dom}\, release\_snd \cup \mathrm{dom}\, release\_rcv$
$state(self?) = running$
$topWaiting! \in \mathrm{dom}\, time$
$\forall wt : \mathrm{dom}\, time \bullet time(topWaiting!) \leq time(wt)$
$\forall wt : \mathrm{dom}\, time \mid time(wt) = time(topWaiting!)$
$\quad \bullet priority(topWaiting!) \geq priority(wt)$
$priority(topWaiting!)$
$\quad > priority(running\_tasks(executable(topWaiting!)))$
$\exists st? : STATE \mid st? = ready$
$\quad \bullet Reschedule[topWaiting!/target?, tasks/tasks?,$
$\quad\quad executable/executable?, priority/pri?]$
$\Xi QueueData$
$wait\_snd' = \{topWaiting!\} \lhd wait\_snd$
$wait\_rcv' = \{topWaiting!\} \lhd wait\_rcv$
$\Xi QReleasingData$
$clock' = time(topWaiting!)$
$delayed\_task' = delayed\_task \setminus \{topWaiting!\}$
$time' = \{topWaiting!\} \lhd time$
$time\_slice' = time\_slice$

$CheckDelayedTask\_TQT \mathrel{\widehat=} CheckDelayedTaskN\_TQT$
$\qquad\qquad\qquad\qquad \lor\ CheckDelayedTaskS\_TQT$

TimeSlicing_TQT

$\Delta TaskQueueTime$
$topReadys! : \mathbb{F}\ TASK$

$\#topReadys! \leq \#cores$
$\forall t : topReadys! \bullet state(t) = ready$
$\quad \land priority(t) = priority(running\_tasks(executable(t)))$
$\forall t1, t2 : topReadys! \mid executable(t1) = executable(t2) \bullet t1 = t2$
$\forall c : cores \mid (\forall t : topReadys! \bullet executable(t) \neq c)$
$\quad \bullet (\forall t : executable^{\sim}(\!|\ \{c\}\ |\!) \mid state(t) = ready$
$\quad\quad \bullet priority(t) < priority(running\_tasks(c)))$
$topReadys! \neq \emptyset$
$\forall t : \mathrm{dom}\, time \bullet time\_slice \leq time(t)$

$$
\begin{array}{l}
tasks' = tasks \\
executable(\!| \ topReadys! \ |\!) \lhd running\_tasks' \\
\quad = executable(\!| \ topReadys! \ |\!) \lhd running\_tasks \\
executable' = executable \\
(running\_tasks(\!| \ executable(\!| \ topReadys! \ |\!) \ |\!) \cup topReadys!) \lhd state' \\
\quad = (running\_tasks(\!| \ executable(\!| \ topReadys! \ |\!) \ |\!) \cup topReadys!) \lhd state \\
executable(\!| \ topReadys! \ |\!) \lhd phys\_context' \\
\quad = executable(\!| \ topReadys! \ |\!) \lhd phys\_context \\
running\_tasks(\!| \ executable(\!| \ topReadys! \ |\!) \ |\!) \lhd log\_context' \\
\quad = running\_tasks(\!| \ executable(\!| \ topReadys! \ |\!) \ |\!) \lhd log\_context \\
priority' = priority \\
\forall \, trt : topReadys! \\
\quad \bullet \ running\_tasks'(executable(trt)) = trt \\
\qquad \land \ state'(trt) = running \\
\qquad \land \ state'(running\_tasks(executable(trt))) = ready \\
\qquad \land \ phys\_context'(executable(trt)) = log\_context(trt) \\
\qquad \land \ log\_context'(running\_tasks(executable(trt))) \\
\qquad\quad = phys\_context(executable(trt)) \\
\Xi Queue \\
clock' = clock \\
delayed\_task' = delayed\_task \\
time' = time \\
time\_slice' = time\_slice + slice\_delay
\end{array}
$$

NoSlicing_TQT

$$
\begin{array}{l}
\Delta TaskQueueTime \\
\hline
\forall \, c : cores \\
\quad \bullet \ \forall \, rt : state^{\sim}(\!| \ \{ready\} \ |\!) \mid executable(rt) = c \\
\qquad \bullet \ priority(rt) < priority(running\_tasks(c)) \\
\forall \, t : \mathrm{dom} \, time \bullet time\_slice \leq time(t) \\
\Xi TaskQueue \\
clock' = clock \\
delayed\_task' = delayed\_task \\
time' = time \\
time\_slice' = time\_slice + slice\_delay
\end{array}
$$

Appendix K

SPECIFICATION FOR MULTI-CORE MUTEX MODEL

MutexData

$$semaphore : \mathbb{P}\, QUEUE$$
$$mutex : \mathbb{P}\, QUEUE$$
$$mutex\_holder : QUEUE \pfun TASK$$
$$mutex\_recursive : QUEUE \pfun \mathbb{N}$$

$$mutex \cap semaphore = \emptyset$$
$$\mathrm{dom}\, mutex\_recursive = mutex$$
$$\forall\, m : mutex \bullet m \notin \mathrm{dom}\, mutex\_holder \Leftrightarrow mutex\_recursive(m) = 0$$

Init_MutexData

$$MutexData'$$

$$semaphore' = \emptyset$$
$$mutex' = \emptyset$$
$$mutex\_holder' = \emptyset$$
$$mutex\_recursive' = \emptyset$$

OriginalPrioData

$$base\_priority : TASK \pfun \mathbb{N}$$

Init_OriginalPrioData

$$OriginalPrioData'$$

$base\_priority' = \emptyset$

MReleasingData

$release\_mutex : TASK \pfun QUEUE$

Init_MReleasingData

$MReleasingData'$

$release\_mutex' = \emptyset$

Mutex

$MutexData$
$OriginalPrioData$
$MReleasingData$

$\dom base\_priority = \ran mutex\_holder$
$\ran release\_mutex \subseteq mutex$

Init_Mutex

$Mutex'$

$Init\_MutexData$
$Init\_OriginalPrioData$
$Init\_MReleasingData$

TaskQueueTimeMutex

$TaskQueueTime$
$Mutex$

$semaphore \subseteq queue$
$\forall s : semaphore \bullet q\_max(s) = 1$
$mutex \subseteq queue$
$\forall m : mutex \bullet q\_max(m) = 1$
$\dom mutex\_holder = \{ m : mutex \mid q\_size(m) = 0 \}$
$\forall m : \dom mutex\_holder \bullet executable(mutex\_holder(m)) \in q\_ava(m)$
$\forall mh : \ran mutex\_holder \bullet priority(mh) \geq base\_priority(mh)$
$\forall ms : mutex \cup semaphore \bullet ms \notin \ran wait\_snd \cup \ran release\_snd$
$release\_mutex \subseteq release\_rcv$

Init

$TaskQueueTimeMutex'$

$$Init\_TaskQueueTime$$
$$Init\_Mutex$$

ExtendTQTXi

$$\Delta TaskQueueTimeMutex$$

$$\Xi Mutex$$

$$CreateTask\_TQTM \mathrel{\widehat=} ExtendTQTXi \land CreateTask\_TQT$$

DeleteTask_TQTM

$$DeleteTask\_TQT$$
$$\Delta TaskQueueTimeMutex$$

$$target? \notin \operatorname{ran} mutex\_holder$$
$$\Xi MutexData$$
$$\Xi OriginalPrioData$$
$$release\_mutex' = \{target?\} \lhd release\_mutex$$

$$ExecuteRunningTask\_TQTM \mathrel{\widehat=} ExtendTQTXi \land ExecuteRunningTask\_TQT$$

$$SuspendTask\_TQTM \mathrel{\widehat=} ExtendTQTXi \land SuspendTask\_TQT$$

$$ResumeTask\_TQTM \mathrel{\widehat=} ExtendTQTXi \land ResumeTask\_TQT$$

ChangeTaskPriorityNotHolder_TQTM

$$ChangeTaskPriority\_TQT$$
$$\Delta TaskQueueTimeMutex$$

$$target? \notin \operatorname{dom} base\_priority$$
$$\Xi Mutex$$

ChangeTaskPriorityNotInherited_TQTM

$$ChangeTaskPriority\_TQT$$
$$\Delta TaskQueueTimeMutex$$

$$target? \in \operatorname{dom} base\_priority$$
$$base\_priority(target?) = priority(target?)$$
$$\Xi MutexData$$
$$base\_priority' = base\_priority \oplus \{(target? \mapsto newpri?)\}$$

$\Xi MReleasingData$

ChangeTaskPriorityInheritedN_TQTM

$\Delta TaskQueueTimeMutex$
$newpri? : \mathbb{N}$
$target? : TASK$
$self? : TASK$
$topReady! : TASK$

$self? \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv$
$state(self?) = running$
$target? \in \operatorname{dom} base\_priority$
$base\_priority(target?) \neq priority(target?)$
$state(target?) \neq nonexistent$
$newpri? \leq priority(target?)$
$target? \in idles \Rightarrow newpri? = 0$
$\Xi TaskQueueTime$
$\Xi MutexData$
$base\_priority' = base\_priority \oplus \{(target? \mapsto newpri?)\}$
$\Xi MReleasingData$
$topReady! = self?$

ChangeTaskPriorityInheritedU_TQTM

$ChangeTaskPriorityN\_T$
$\Delta TaskQueueTimeMutex$

$target? \in \operatorname{dom} base\_priority$
$base\_priority(target?) \neq priority(target?)$
$newpri? > priority(target?)$
$\Xi Queue$
$\Xi Time$
$\Xi MutexData$
$base\_priority' = base\_priority \oplus \{(target? \mapsto newpri?)\}$
$\Xi MReleasingData$

ChangeTaskPriorityInheritedS_TQTM

$ChangeTaskPriorityS\_T$
$\Delta TaskQueueTimeMutex$

$target? \in \operatorname{dom} base\_priority$
$base\_priority(target?) \neq priority(target?)$
$newpri? > priority(target?)$
$\Xi Queue$
$\Xi Time$

$$\Xi MutexData$$
$$base\_priority' = base\_priority \oplus \{(target? \mapsto newpri?)\}$$
$$\Xi MReleasingData$$

$$ChangeTaskPriority\_TQTM \mathrel{\widehat=} ChangeTaskPriorityNotHolder\_TQTM$$
$$\lor\ ChangeTaskPriorityNotInherited\_TQTM$$
$$\lor\ ChangeTaskPriorityInheritedN\_TQTM$$
$$\lor\ ChangeTaskPriorityInheritedU\_TQTM$$
$$\lor\ ChangeTaskPriorityInheritedS\_TQTM$$

$$Migration\_TQTM \mathrel{\widehat=} ExtendTQTXi \land Migration\_TQT$$

$$CreateQueue\_TQTM \mathrel{\widehat=} ExtendTQTXi \land CreateQueue\_TQT$$

DeleteQueue_TQTM

$$DeleteQueue\_TQT$$
$$\Delta TaskQueueTimeMutex$$

$$que? \notin semaphore \cup mutex$$
$$\Xi Mutex$$

QueueSend_TQTM

$$QueueSend\_TQT$$
$$\Delta TaskQueueTimeMutex$$

$$que? \notin mutex \cup mutex$$
$$\Xi Mutex$$

QueueReceive_TQTM

$$QueueReceive\_TQT$$
$$\Delta TaskQueueTimeMutex$$

$$self? \notin \mathrm{dom}\, release\_mutex$$
$$que? \notin mutex \cup semaphore$$
$$\Xi Mutex$$

$$ChangeQueueLevel\_TQTM \mathrel{\widehat=} ExtendTQTXi \land ChangeQueueLevel\_TQT$$

$$DelayUntil\_TQTM \mathrel{\widehat=} ExtendTQTXi \land DelayUntil\_TQT$$

$$CheckDelayedTask\_TQTM \mathrel{\widehat=} ExtendTQTXi \land CheckDelayedTask\_TQT$$

$$TimeSlicing\_TQTM \mathrel{\widehat=} ExtendTQTXi \land TimeSlicing\_TQT$$

$$NoSlicing\_TQTM \mathrel{\widehat=} ExtendTQTXi \land NoSlicing\_TQT$$

CreateBinarySemaphore_TQTM

$\Delta TaskQueueTimeMutex$
$sem? : QUEUE$
$self? : TASK$
$cset? : \mathbb{F}\, cores$

$self? \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv$
$state(self?) = running$
$sem? \notin queue$
$cset? \neq \emptyset$
$\Xi Task$
$queue' = queue \cup \{sem?\}$
$q\_max' = q\_max \oplus \{(sem? \mapsto 1)\}$
$q\_size' = q\_size \oplus \{(sem? \mapsto 1)\}$
$q\_ava' = q\_ava \oplus \{(sem? \mapsto cset?)\}$
$\Xi WaitingData$
$\Xi QReleasingData$
$\Xi Time$
$semaphore' = semaphore \cup \{sem?\}$
$mutex' = mutex$
$mutex\_holder' = mutex\_holder$
$mutex\_recursive' = mutex\_recursive$
$\Xi OriginalPrioData$
$\Xi MReleasingData$

DeleteBinarySemaphore_TQTM

$\Delta TaskQueueTimeMutex$
$sem? : QUEUE$
$self? : TASK$

$sem? \in semaphore$
$DeleteQueue\_TQT[sem?/que?]$
$semaphore' = semaphore \setminus \{sem?\}$
$mutex' = mutex$
$mutex\_holder' = mutex\_holder$
$mutex\_recursive' = mutex\_recursive$
$\Xi OriginalPrioData$
$\Xi MReleasingData$

CreateMutex_TQTM

$\Delta TaskQueueTimeMutex$
$mut? : QUEUE$
$self? : TASK$
$cset? : \mathbb{F}\ cores$

$self? \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv$
$state(self?) = running$
$mut? \notin queue$
$cset? \neq \emptyset$
$\Xi Task$
$queue' = queue \cup \{mut?\}$
$q\_max' = q\_max \oplus \{(mut? \mapsto 1)\}$
$q\_size' = q\_size \oplus \{(mut? \mapsto 1)\}$
$q\_ava' = q\_ava \oplus \{(mut? \mapsto cset?)\}$
$\Xi WaitingData$
$\Xi Time$
$semaphore' = semaphore$
$mutex' = mutex \cup \{mut?\}$
$mutex\_holder' = mutex\_holder$
$mutex\_recursive' = mutex\_recursive \oplus \{(mut? \mapsto 0)\}$
$\Xi OriginalPrioData$
$\Xi MReleasingData$

DeleteMutex_TQTM

$\Delta TaskQueueTimeMutex$
$mut? : QUEUE$
$self? : TASK$

$mut? \in mutex \setminus \operatorname{dom} mutex\_holder$
$DeleteQueue\_TQT[mut?/que?]$
$semaphore' = semaphore$
$mutex' = mutex \setminus \{mut?\}$
$mutex\_holder' = mutex\_holder$
$mutex\_recursive' = \{mut?\} \lhd mutex\_recursive$
$\Xi OriginalPrioData$
$\Xi MReleasingData$

MutexTakeNnonInh_TQTM

$\Delta TaskQueueTimeMutex$
$mut? : QUEUE$
$self? : TASK$
$topReady! : TASK$

$self? \in \operatorname{dom} release\_rcv \Rightarrow self? \in \operatorname{dom} release\_mutex$

$mut? \in mutex$
$self? \notin \mathrm{dom}\ base\_priority$
$QueueReceiveN\_TQT[mut?/que?]$
$semaphore' = semaphore$
$mutex' = mutex$
$mutex\_holder' = mutex\_holder \oplus \{(mut? \mapsto self?)\}$
$mutex\_recursive' = mutex\_recursive \oplus$
$\quad \{(mut? \mapsto mutex\_recursive(mut?) + 1)\}$
$base\_priority' = base\_priority \oplus \{(self? \mapsto priority(self?))\}$
$release\_mutex' = \{self?\} \lhd release\_mutex$

MutexTakeNInh_TQTM

$\Delta TaskQueueTimeMutex$
$mut? : QUEUE$
$self? : TASK$
$topReady! : TASK$

$self? \in \mathrm{dom}\ release\_rcv \Rightarrow self? \in \mathrm{dom}\ release\_mutex$
$mut? \in mutex$
$self? \in \mathrm{dom}\ base\_priority$
$QueueReceiveN\_TQT[mut?/que?]$
$semaphore' = semaphore$
$mutex' = mutex$
$mutex\_holder' = mutex\_holder \oplus \{(mut? \mapsto self?)\}$
$mutex\_recursive' = mutex\_recursive \oplus$
$\quad \{(mut? \mapsto mutex\_recursive(mut?) + 1)\}$
$\Xi OriginalPrioData$
$release\_mutex' = \{self?\} \lhd release\_mutex$

MutexTakeRecursive_TQTM

$\Delta TaskQueueTimeMutex$
$mut? : QUEUE$
$self? : TASK$
$topReady! : TASK$

$self? \notin \mathrm{dom}\ release\_snd \cup \mathrm{dom}\ release\_rcv$
$mut? \in mutex$
$state(self?) = running$
$executable(self?) \in q\_ava(mut?)$
$self? = mutex\_holder(mut?)$
$\Xi TaskQueueTime$
$semaphore' = semaphore$
$mutex' = mutex$
$mutex\_holder' = mutex\_holder$
$mutex\_recursive' = mutex\_recursive \oplus$

$\{(mut? \mapsto mutex\_recursive(mut?) + 1)\}$
$\Xi OriginalPrioData$
$\Xi MReleasingData$
$topReady! = self?$

MutexTakeEnonInh_TQTM

$\Delta TaskQueueTimeMutex$
$mut? : QUEUE$
$self? : TASK$
$topReady! : TASK$
$n? : \mathbb{N}$

$self? \in \operatorname{dom} release\_rcv \Rightarrow self? \in \operatorname{dom} release\_mutex$
$mut? \in \operatorname{dom} mutex\_holder$
$priority(self?) \leq priority(mutex\_holder(mut?))$
$self? \neq mutex\_holder(mut?)$
$QueueReceiveE\_TQT[mut?/que?]$
$\Xi MutexData$
$\Xi OriginalPrioData$
$release\_mutex' = \{self?\} \lhd release\_mutex$

MutexTakeEInheritSameCoreHolder_TQTM

$\Delta TaskQueueTimeMutex$
$mut? : QUEUE$
$self? : TASK$
$topReady! : TASK$
$n? : \mathbb{N}$

$self? \notin \operatorname{dom} release\_snd$
$self? \in \operatorname{dom} release\_rcv \Rightarrow mut? = release\_rcv(self?)$
$self? \in \operatorname{dom} release\_rcv \Rightarrow self? \in \operatorname{dom} release\_mutex$
$mut? \in \operatorname{dom} mutex\_holder$
$state(self?) = running$
$executable(self?) \in q\_ava(mut?)$
$priority(self?) > priority(mutex\_holder(mut?))$
$n? > clock$
$executable(mutex\_holder(mut?)) = executable(self?)$
$mutex\_holder(mut?) \in state^{\sim}(\!|\ \{ready\}\ |\!)$
$topReady! = mutex\_holder(mut?)$
$\exists\, st? : STATE;\ pri? : TASK \to \mathbb{N}$
$\quad | \ st? = blocked$
$\quad\quad \wedge\ pri? = priority \oplus \{(topReady! \mapsto priority(self?))\}$
$\quad \bullet\ Reschedule[topReady!/target?, tasks/tasks?, executable/executable?]$
$\Xi QueueData$
$wait\_snd' = wait\_snd$

$wait\_rcv' = wait\_rcv \oplus \{(self? \mapsto mut?)\}$
$release\_snd' = release\_snd$
$release\_rcv' = \{self?\} \lhd release\_rcv$
$clock' = clock$
$delayed\_task' = delayed\_task$
$time' = time \oplus \{(self? \mapsto n?)\}$
$time\_slice' = time\_slice$
$\Xi MutexData$
$\Xi OriginalPrioData$
$release\_mutex' = \{self?\} \lhd release\_mutex$

MutexTakeEInheritSameCoreReady_TQTM

$\Delta TaskQueueTimeMutex$
$mut? : QUEUE$
$self? : TASK$
$topReady! : TASK$
$n? : \mathbb{N}$

$self? \notin \mathrm{dom}\, release\_snd$
$self? \in \mathrm{dom}\, release\_rcv \Rightarrow mut? = release\_rcv(self?)$
$self? \in \mathrm{dom}\, release\_rcv \Rightarrow self? \in \mathrm{dom}\, release\_mutex$
$mut? \in \mathrm{dom}\, mutex\_holder$
$state(self?) = running$
$executable(self?) \in q\_ava(mut?)$
$priority(self?) > priority(mutex\_holder(mut?))$
$n? > clock$
$executable(mutex\_holder(mut?)) = executable(self?)$
$mutex\_holder(mut?) \notin state^{\sim}(\!| \{ready\} |\!)$
$findTopReady[self?/target?]$
$\exists st? : STATE;\ pri? : TASK \to \mathbb{N}$
$\quad | \ st? = blocked$
$\quad\quad \wedge\ pri? = priority \oplus \{(mutex\_holder(mut?) \mapsto priority(self?))\}$
$\quad \bullet\ Reschedule[topReady!/target?, tasks/tasks?, executable/executable?]$
$\Xi QueueData$
$wait\_snd' = wait\_snd$
$wait\_rcv' = wait\_rcv \oplus \{(self? \mapsto mut?)\}$
$release\_snd' = release\_snd$
$release\_rcv' = \{self?\} \lhd release\_rcv$
$clock' = clock$
$delayed\_task' = delayed\_task$
$time' = time \oplus \{(self? \mapsto n?)\}$
$time\_slice' = time\_slice$
$\Xi MutexData$
$\Xi OriginalPrioData$
$release\_mutex' = \{self?\} \lhd release\_mutex$

MutexTakeEInheritDiffCoreN_TQTM

$\Delta TaskQueueTimeMutex$
$mut? : QUEUE$
$self? : TASK$
$topReady! : TASK$
$n? : \mathbb{N}$

$self? \notin \operatorname{dom} release\_snd$
$self? \in \operatorname{dom} release\_rcv \Rightarrow mut? = release\_rcv(self?)$
$self? \in \operatorname{dom} release\_rcv \Rightarrow self? \in \operatorname{dom} release\_mutex$
$mut? \in \operatorname{dom} mutex\_holder$
$state(self?) = running$
$executable(self?) \in q\_ava(mut?)$
$priority(self?) > priority(mutex\_holder(mut?))$
$n? > clock$
$executable(mutex\_holder(mut?)) \neq executable(self?)$
$priority(self?) >$
$\quad priority(running\_tasks(executable(mutex\_holder(mut?))))$
$\quad\quad \Rightarrow mutex\_holder(mut?) \notin state^{\sim}(\!|\ \{ready\}\ |\!)$
$findTopReady[self?/target?]$
$tasks' = tasks$
$running\_tasks' = running\_tasks \oplus \{(executable(self?) \mapsto topReady!)\}$
$executable' = executable$
$state' = state \oplus \{(self? \mapsto blocked),$
$\quad (topReady! \mapsto running),$
$\quad (mutex\_holder(mut?) \mapsto ready)\}$
$phys\_context' = phys\_context \oplus$
$\quad \{(executable(self?) \mapsto log\_context(topReady!))\}$
$log\_context' = log\_context \oplus \{(self? \mapsto phys\_context(executable(self?)))\}$
$priority' = priority \oplus \{(mutex\_holder(mut?) \mapsto priority(self?))\}$
$\Xi QueueData$
$wait\_snd' = wait\_snd$
$wait\_rcv' = wait\_rcv \oplus \{(self? \mapsto mut?)\}$
$release\_snd' = release\_snd$
$release\_rcv' = \{self?\} \lhd release\_rcv$
$clock' = clock$
$delayed\_task' = delayed\_task$
$time' = time \oplus \{(self? \mapsto n?)\}$
$time\_slice' = time\_slice$
$\Xi MutexData$
$\Xi OriginalPrioData$
$release\_mutex' = \{self?\} \lhd release\_mutex$

MutexTakeEInheritDiffCoreS_TQTM

$\Delta TaskQueueTimeMutex$
$mut? : QUEUE$

$$
\begin{array}{|l}
self? : TASK \\
topReady! : TASK \\
n? : \mathbb{N} \\
\hline
self? \notin \mathrm{dom}\, release\_snd \\
self? \in \mathrm{dom}\, release\_rcv \Rightarrow mut? = release\_rcv(self?) \\
self? \in \mathrm{dom}\, release\_rcv \Rightarrow self? \in \mathrm{dom}\, release\_mutex \\
mut? \in \mathrm{dom}\, mutex\_holder \\
state(self?) = running \\
executable(self?) \in q\_ava(mut?) \\
priority(self?) > priority(mutex\_holder(mut?)) \\
n? > clock \\
executable(mutex\_holder(mut?)) \neq executable(self?) \\
priority(self?) > \\
\quad priority(running\_tasks(executable(mutex\_holder(mut?)))) \\
mutex\_holder(mut?) \in state^{\sim}(\!| \{ready\} |\!) \\
findTopReady[self?/target?] \\
tasks' = tasks \\
running\_tasks' = running\_tasks \oplus \{(executable(self?) \mapsto topReady!), \\
\quad (executable(mutex\_holder(mut?)) \mapsto mutex\_holder(mut?))\} \\
executable' = executable \\
state' = state \oplus \{(self? \mapsto blocked), \\
\quad (topReady! \mapsto running), \\
\quad (running\_tasks(executable(mutex\_holder(mut?))) \mapsto ready), \\
\quad (mutex\_holder(mut?) \mapsto running)\} \\
phys\_context' = phys\_context \oplus \\
\quad \{(executable(self?) \mapsto log\_context(topReady!)), \\
\qquad (executable(mutex\_holder(mut?)) \mapsto \\
\qquad\quad log\_context(mutex\_holder(mut?)))\} \\
log\_context' = log\_context \oplus \{(self? \mapsto phys\_context(executable(self?))), \\
\quad (running\_tasks(executable(mutex\_holder(mut?))) \\
\qquad \mapsto phys\_context(executable(mutex\_holder(mut?))))\} \\
priority' = priority \oplus \{(mutex\_holder(mut?) \mapsto priority(self?))\} \\
\Xi QueueData \\
wait\_snd' = wait\_snd \\
wait\_rcv' = wait\_rcv \oplus \{(self? \mapsto mut?)\} \\
release\_snd' = release\_snd \\
release\_rcv' = \{self?\} \lhd release\_rcv \\
clock' = clock \\
delayed\_task' = delayed\_task \\
time' = time \oplus \{(self? \mapsto n?)\} \\
time\_slice' = time\_slice \\
\Xi MutexData \\
\Xi OriginalPrioData \\
release\_mutex' = \{self?\} \lhd release\_mutex \\
\hline
\end{array}
$$

basePriorityMan

$\Delta TaskQueueTimeMutex$
$mut? : QUEUE$
$self? : TASK$

$self? \in \mathrm{ran}(\{mut?\} \lhd mutex\_holder) \Rightarrow \Xi OriginalPrioData$
$self? \notin \mathrm{ran}(\{mut?\} \lhd mutex\_holder)$
$\quad \Rightarrow base\_priority' = \{self?\} \lhd base\_priority$

MutexGiveNRecursive_TQTM

$\Delta TaskQueueTimeMutex$
$mut? : QUEUE$
$self? : TASK$
$topReady! : TASK$

$self? \notin \mathrm{dom}\, release\_snd \cup \mathrm{dom}\, release\_rcv$
$mut? \in \mathrm{dom}\, mutex\_holder$
$self? = mutex\_holder(mut?)$
$state(self?) = running$
$mutex\_recursive(mut?) > 1$
$\Xi TaskQueueTime$
$semaphore' = semaphore$
$mutex' = mutex$
$mutex\_holder' = mutex\_holder$
$mutex\_recursive' = mutex\_recursive \oplus$
$\quad \{(mut? \mapsto mutex\_recursive(mut?) - 1)\}$
$\Xi OriginalPrioData$
$\Xi MReleasingData$
$topReady! = self?$

MutexGiveNnonInh_TQTM

$\Delta TaskQueueTimeMutex$
$mut? : QUEUE$
$self? : TASK$
$topReady! : TASK$

$self? \notin \mathrm{dom}\, release\_snd \cup \mathrm{dom}\, release\_rcv$
$mut? \in \mathrm{dom}\, mutex\_holder$
$self? = mutex\_holder(mut?)$
$mutex\_recursive(mut?) = 1$
$base\_priority(self?) = priority(self?)$
$QueueSendN\_TQT[mut?/que?]$
$semaphore' = semaphore$
$mutex' = mutex$
$mutex\_holder' = \{mut?\} \lhd mutex\_holder$

$mutex\_recursive' = mutex\_recursive \oplus \{(mut? \mapsto 0)\}$
$basePriorityMan$
$\Xi MReleasingData$

MutexGiveNInhN_TQTM

$\Delta TaskQueueTimeMutex$
$mut? : QUEUE$
$self? : TASK$
$topReady! : TASK$

$self? \notin \mathrm{dom}\, release\_snd \cup \mathrm{dom}\, release\_rcv$
$mut? \in \mathrm{dom}\, mutex\_holder$
$self? = mutex\_holder(mut?)$
$state(self?) = running$
$mutex\_recursive(mut?) = 1$
$mut? \notin \mathrm{ran}\, wait\_rcv$
$base\_priority(self?) \neq priority(self?)$
$\forall rt : state^{\sim}(\!| \{ready\} |\!) \mid executable(rt) = executable(self?)$
$\quad \bullet\ base\_priority(self?) \geq priority(rt)$
$\Xi TaskData$
$\Xi StateData$
$\Xi ContextData$
$priority' = priority \oplus \{(self? \mapsto base\_priority(self?))\}$
$queue' = queue$
$q\_max' = q\_max$
$q\_size' = q\_size \oplus \{(mut? \mapsto 1)\}$
$q\_ava' = q\_ava$
$\Xi WaitingData$
$\Xi QReleasingData$
$semaphore' = semaphore$
$mutex' = mutex$
$mutex\_holder' = \{mut?\} \lhd mutex\_holder$
$mutex\_recursive' = mutex\_recursive \oplus \{(mut? \mapsto 0)\}$
$basePriorityMan$
$\Xi MReleasingData$
$topReady! = self?$

MutexGiveNInhS_TQTM

$\Delta TaskQueueTimeMutex$
$mut? : QUEUE$
$self? : TASK$
$topReady! : TASK$

$self? \notin \mathrm{dom}\, release\_snd \cup \mathrm{dom}\, release\_rcv$
$mut? \in \mathrm{dom}\, mutex\_holder$

$self? = mutex\_holder(mut?)$
$state(self?) = running$
$mutex\_recursive(mut?) = 1$
$mut? \notin \operatorname{ran} wait\_rcv$
$base\_priority(self?) \neq priority(self?)$
$findTopReady[self?/target?]$
$base\_priority(self?) < priority(topReady!)$
$\exists\, st? : STATE;\ pri? : TASK \to \mathbb{N}$
$\quad \mid st? = ready \land pri? = priority \oplus \{(self? \mapsto base\_priority(self?))\}$
$\quad \bullet\ Reschedule[topReady!/target?, tasks/tasks?, executable/executable?]$
$queue' = queue$
$q\_max' = q\_max$
$q\_size' = q\_size \oplus \{(mut? \mapsto 1)\}$
$q\_ava' = q\_ava$
$\Xi WaitingData$
$\Xi QReleasingData$
$semaphore' = semaphore$
$mutex' = mutex$
$mutex\_holder' = \{mut?\} \lhd mutex\_holder$
$mutex\_recursive' = mutex\_recursive \oplus \{(mut? \mapsto 0)\}$
$basePriorityMan$
$\Xi MReleasingData$

MutexGiveWnonInhN_TQTM

$\Delta TaskQueueTimeMutex$
$mut? : QUEUE$
$self? : TASK$
$topReady! : TASK$

$self? \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv$
$mut? \in \operatorname{dom} mutex\_holder$
$self? = mutex\_holder(mut?)$
$mutex\_recursive(mut?) = 1$
$base\_priority(self?) = priority(self?)$
$QueueSendW\_TQT[mut?/que?]$
$semaphore' = semaphore$
$mutex' = mutex$
$mutex\_holder' = \{mut?\} \lhd mutex\_holder$
$mutex\_recursive' = mutex\_recursive \oplus \{(mut? \mapsto 0)\}$
$basePriorityMan$
$release\_mutex' = release\_mutex \oplus \{(topReady! \mapsto mut?)\}$

MutexGiveWnonInhS_TQTM

$\Delta TaskQueueTimeMutex$

$mut? : QUEUE$
$self? : TASK$
$topReady! : TASK$

$self? \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv$
$mut? \in \operatorname{dom} mutex\_holder$
$self? = mutex\_holder(mut?)$
$mutex\_recursive(mut?) = 1$
$base\_priority(self?) = priority(self?)$
$QueueSendWS\_TQT[mut?/que?]$
$semaphore' = semaphore$
$mutex' = mutex$
$mutex\_holder' = \{mut?\} \lhd mutex\_holder$
$mutex\_recursive' = mutex\_recursive \oplus \{(mut? \mapsto 0)\}$
$basePriorityMan$
$release\_mutex' = release\_mutex \oplus \{(topReady! \mapsto mut?)\}$

MutexGiveWInhN_TQTM

$\Delta TaskQueueTimeMutex$
$mut? : QUEUE$
$self? : TASK$
$topReady! : TASK$

$self? \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv$
$mut? \in \operatorname{dom} mutex\_holder$
$self? = mutex\_holder(mut?)$
$state(self?) = running$
$mutex\_recursive(mut?) = 1$
$topReady! \in wait\_rcv^{\sim}(\!| \{mut?\} |\!)$
$\forall wr : wait\_rcv^{\sim}(\!| \{mut?\} |\!) \bullet priority(topReady!) \geq priority(wr)$
$priority(topReady!) \leq priority(running\_tasks(executable(topReady!)))$
$base\_priority(self?) \neq priority(self?)$
$\forall rt : state^{\sim}(\!| \{ready\} |\!) \mid executable(self?) = executable(rt)$
$\quad \bullet base\_priority(self?) \geq priority(rt)$
$executable(self?) = executable(topReady!)$
$\quad \Rightarrow priority(topReady!) \leq base\_priority(self?)$
$\Xi TaskData$
$state' = state \oplus \{(topReady! \mapsto ready)\}$
$\Xi ContextData$
$priority' = priority \oplus \{(self? \mapsto base\_priority(self?))\}$
$queue' = queue$
$q\_max' = q\_max$
$q\_size' = q\_size \oplus \{(mut? \mapsto 1)\}$
$q\_ava' = q\_ava$
$wait\_snd' = wait\_snd$
$wait\_rcv' = \{topReady!\} \lhd wait\_rcv$

$release\_snd' = release\_snd$
$release\_rcv' = release\_rcv \oplus \{(topReady! \mapsto mut?)\}$
$clock' = clock$
$delayed\_task' = delayed\_task$
$time' = \{topReady!\} \lhd time$
$time\_slice' = time\_slice$
$semaphore' = semaphore$
$mutex' = mutex$
$mutex\_holder' = \{mut?\} \lhd mutex\_holder$
$mutex\_recursive' = mutex\_recursive \oplus \{(mut? \mapsto 0)\}$
$basePriorityMan$
$release\_mutex' = release\_mutex \oplus \{(topReady! \mapsto mut?)\}$

MutexGiveWInhSR_TQTM

$\Delta TaskQueueTimeMutex$
$mut? : QUEUE$
$self? : TASK$
$topReady! : TASK$
$topWaiting! : TASK$

$self? \notin \mathrm{dom}\, release\_snd \cup \mathrm{dom}\, release\_rcv$
$mut? \in \mathrm{dom}\, mutex\_holder$
$self? = mutex\_holder(mut?)$
$state(self?) = running$
$mutex\_recursive(mut?) = 1$
$topWaiting! \in wait\_rcv^{\sim}(\!|\ \{mut?\}\ |\!)$
$\forall\, wr : wait\_rcv^{\sim}(\!|\ \{mut?\}\ |\!) \bullet priority(topWaiting!) \geq priority(wr)$
$findTopReady[self?/target?]$
$base\_priority(self?) \neq priority(self?)$
$executable(topWaiting!) \neq executable(self?)$
$\quad \Rightarrow (priority(topWaiting!)$
$\quad\quad \leq priority(running\_tasks(executable(topWaiting!)))$
$\quad\quad\quad \land\ base\_priority(self?) < priority(topReady!))$
$executable(topWaiting!) = executable(self?)$
$\quad \Rightarrow (priority(topReady!) > base\_priority(self?)$
$\quad\quad \land\ priority(topReady!) > priority(topWaiting!))$
$tasks' = tasks$
$running\_tasks' = running\_tasks \oplus \{(executable(topReady!) \mapsto topReady!)\}$
$executable' = executable$
$state' = state \oplus$
$\quad \{(self? \mapsto ready), (topReady! \mapsto running), (topWaiting! \mapsto ready)\}$
$phys\_context' = phys\_context \oplus$
$\quad \{(executable(topReady!) \mapsto log\_context(topReady!))\}$
$log\_context' = log\_context \oplus$
$\quad \{(self? \mapsto phys\_context(executable(topReady!)))\}$
$priority' = priority \oplus \{(self? \mapsto base\_priority(self?))\}$

$$
\begin{array}{l}
queue' = queue \\
q\_max' = q\_max \\
q\_size' = q\_size \oplus \{(mut? \mapsto 1)\} \\
q\_ava' = q\_ava \\
wait\_snd' = wait\_snd \\
wait\_rcv' = \{topWaiting!\} \lhd wait\_rcv \\
release\_snd' = release\_snd \\
release\_rcv' = release\_rcv \oplus \{(topWaiting! \mapsto mut?)\} \\
clock' = clock \\
delayed\_task' = delayed\_task \\
time' = \{topWaiting!\} \lhd time \\
time\_slice' = time\_slice \\
semaphore' = semaphore \\
mutex' = mutex \\
mutex\_holder' = \{mut?\} \lhd mutex\_holder \\
mutex\_recursive' = mutex\_recursive \oplus \{(mut? \mapsto 0)\} \\
basePriorityMan \\
release\_mutex' = release\_mutex \oplus \{(topWaiting! \mapsto mut?)\}
\end{array}
$$

MutexGiveWInhSW_TQTM

$$
\begin{array}{l}
\Delta TaskQueueTimeMutex \\
mut? : QUEUE \\
self? : TASK \\
topReady! : TASK \\
topWaiting! : TASK \\
\hline
self? \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv \\
mut? \in \operatorname{dom} mutex\_holder \\
self? = mutex\_holder(mut?) \\
state(self?) = running \\
mutex\_recursive(mut?) = 1 \\
topWaiting! \in wait\_rcv^{\sim}(\!| \{mut?\} |\!) \\
\forall wr : wait\_rcv^{\sim}(\!| \{mut?\} |\!) \bullet priority(topWaiting!) \geq priority(wr) \\
findTopReady[self?/target?] \\
base\_priority(self?) \neq priority(self?) \\
executable(topWaiting!) \neq executable(self?) \\
\quad \Rightarrow (priority(topWaiting!) \\
\qquad > priority(running\_tasks(executable(topWaiting!))) \\
\qquad\quad \wedge base\_priority(self?) \geq priority(topReady!)) \\
executable(topWaiting!) = executable(self?) \\
\quad \Rightarrow (priority(topWaiting!) > base\_priority(self?) \\
\qquad \wedge priority(topWaiting!) \geq priority(topReady!)) \\
\exists st? : STATE;\ pri? : TASK \rightarrow \mathbb{N} \\
\quad | st? = ready \wedge pri? = priority \oplus \{(self? \mapsto base\_priority(self?))\} \\
\quad \bullet Reschedule[topWaiting!/target?, tasks/tasks?, \\
\qquad executable/executable?]
\end{array}
$$

$queue' = queue$
$q\_max' = q\_max$
$q\_size' = q\_size \oplus \{(mut? \mapsto 1)\}$
$q\_ava' = q\_ava$
$wait\_snd' = wait\_snd$
$wait\_rcv' = \{topWaiting!\} \lhd wait\_rcv$
$release\_snd' = release\_snd$
$release\_rcv' = release\_rcv \oplus \{(topWaiting! \mapsto mut?)\}$
$clock' = clock$
$delayed\_task' = delayed\_task$
$time' = \{topWaiting!\} \lhd time$
$time\_slice' = time\_slice$
$semaphore' = semaphore$
$mutex' = mutex$
$mutex\_holder' = \{mut?\} \lhd mutex\_holder$
$mutex\_recursive' = mutex\_recursive \oplus \{(mut? \mapsto 0)\}$
$basePriorityMan$
$release\_mutex' = release\_mutex \oplus \{(topWaiting! \mapsto mut?)\}$

MutexGiveWInhSBoth_TQTM

$\Delta TaskQueueTimeMutex$
$mut? : QUEUE$
$self? : TASK$
$topReady! : TASK$
$topWaiting! : TASK$

$self? \notin \operatorname{dom} release\_snd \cup \operatorname{dom} release\_rcv$
$mut? \in \operatorname{dom} mutex\_holder$
$self? = mutex\_holder(mut?)$
$state(self?) = running$
$mutex\_recursive(mut?) = 1$
$topWaiting! \in wait\_rcv^{\sim}(\!| \{mut?\} |\!)$
$\forall wr : wait\_rcv^{\sim}(\!| \{mut?\} |\!) \bullet priority(topWaiting!) \geq priority(wr)$
$findTopReady[self?/target?]$
$base\_priority(self?) \neq priority(self?)$
$executable(topWaiting!) \neq executable(self?)$
$priority(topWaiting!) > priority(running\_tasks(executable(topWaiting!)))$
$priority(topReady!) > base\_priority(self?)$
$tasks' = tasks$
$running\_tasks' = running\_tasks \oplus$
$\quad \{(executable(topWaiting!) \mapsto topWaiting!),$
$\quad\quad (executable(topReady!) \mapsto topReady!)\}$
$executable' = executable$
$state' = state \oplus \{(self? \mapsto ready), (topReady! \mapsto running),$
$\quad (topWaiting! \mapsto running),$
$\quad\quad (running\_tasks(executable(topWaiting!)) \mapsto ready)\}$

$$
\begin{array}{l}
phys\_context' = phys\_context \oplus \\
\quad \{(executable(topWaiting!) \mapsto log\_context(topWaiting!)), \\
\quad\quad (executable(topReady!) \mapsto log\_context(topReady!))\} \\
log\_context' = log\_context \oplus \\
\quad \{(running\_tasks(executable(topWaiting!)) \\
\quad\quad \mapsto phys\_context(executable(topWaiting!))), \\
\quad\quad\quad (self? \mapsto phys\_context(executable(self?)))\} \\
priority' = priority \oplus \{(self? \mapsto base\_priority(self?))\} \\
queue' = queue \\
q\_max' = q\_max \\
q\_size' = q\_size \oplus \{(mut? \mapsto 1)\} \\
q\_ava' = q\_ava \\
wait\_snd' = wait\_snd \\
wait\_rcv' = \{topWaiting!\} \lhd wait\_rcv \\
release\_snd' = release\_snd \\
release\_rcv' = release\_rcv \oplus \{(topWaiting! \mapsto mut?)\} \\
clock' = clock \\
delayed\_task' = delayed\_task \\
time' = \{topWaiting!\} \lhd time \\
time\_slice' = time\_slice \\
semaphore' = semaphore \\
mutex' = mutex \\
mutex\_holder' = \{mut?\} \lhd mutex\_holder \\
mutex\_recursive' = mutex\_recursive \oplus \{(mut? \mapsto 0)\} \\
basePriorityMan \\
release\_mutex' = release\_mutex \oplus \{(topWaiting! \mapsto mut?)\}
\end{array}
$$

Appendix L

SPECIFICATION FOR MULTI-CORE TASK MODEL WITH PROMOTION

$$[CONTEXT, TASK, CORE]$$

$$\begin{array}{|l} bare\_context : CONTEXT \\ idles : \mathbb{F}\ TASK \\ cores : \mathbb{F}\ CORE \\ \hline \#cores = \#idles \end{array}$$

$$STATE ::= nonexistent \mid ready \mid blocked \mid suspended \mid running$$

$$\begin{aligned} transition == & (\{blocked\} \times \{nonexistent, ready, running, suspended\}) \\ & \cup (\{nonexistent\} \times \{ready, running\}) \\ & \cup (\{ready\} \times \{nonexistent, running, suspended\}) \\ & \cup (\{running\} \times \{blocked, nonexistent, ready, suspended\}) \\ & \cup (\{suspended\} \times \{nonexistent, ready, running\}) \end{aligned}$$

TaskData

$$\begin{array}{|l} tasks : \mathbb{F}\ TASK \\ running\_task : TASK \\ idle : TASK \\ \hline running\_task \in tasks \\ idle \in tasks \\ idle \in idles \end{array}$$

Init_TaskData

$$TaskData'$$

$$tasks' = \{idle'\}$$
$$running\_task' = idle'$$
$$idle' \in idles$$

StateData

$$state : TASK \to STATE$$

ContextData

$$phys\_context : CONTEXT$$
$$log\_context : TASK \to CONTEXT$$

Init_ContextData

$$ContextData'$$

$$phys\_context' = bare\_context$$
$$log\_context' = (\lambda\, x : TASK \bullet bare\_context)$$

PrioData

$$priority : TASK \to \mathbb{N}$$

$$\forall\, i : idles \bullet priority(i) = 0$$

Init_PrioData

$$PrioData'$$

$$priority' = (\lambda\, x : TASK \bullet 0)$$

Task

$$TaskData$$
$$StateData$$
$$ContextData$$
$$PrioData$$

$$tasks = TASK \setminus (state^{\sim}(\!|\, \{nonexistent\} \,|\!))$$
$$state^{\sim}(\!|\, \{running\} \,|\!) = \{running\_task\}$$
$$\forall\, pt : state^{\sim}(\!|\, \{ready\} \,|\!) \bullet priority(running\_task) \geq priority(pt)$$

$\Delta Task$

$Task$

$Task'$

$\forall\, st : TASK \mid state'(st) \neq state(st)$

$\quad \bullet\ state(st) \mapsto state'(st) \in transition$

$idle' = idle$

$Init\_Task$

$Task'$

$Init\_TaskData$

$state' = (\lambda\, x : TASK \bullet nonexistent) \oplus \{(idle' \mapsto running)\}$

$Init\_ContextData$

$Init\_PrioData$

$Reschedule$

$\Delta Task$

$target? : TASK$

$tasks? : \mathbb{P}\ TASK$

$st? : STATE$

$pri? : TASK \to \mathbb{N}$

$tasks' = tasks?$

$running\_task' = target?$

$state' = state \oplus \{(target? \mapsto running), (running\_task \mapsto st?)\}$

$phys\_context' = log\_context(target?)$

$log\_context' = log\_context \oplus \{(running\_task \mapsto phys\_context)\}$

$priority' = pri?$

$CreateTaskN\_T$

$\Delta Task$

$target? : TASK$

$newpri? : \mathbb{N}$

$state(target?) = nonexistent$

$newpri? \leq priority(running\_task)$

$tasks' = tasks \cup \{target?\}$

$running\_task' = running\_task$

$state' = state \oplus \{(target?, ready)\}$

$\Xi ContextData$

$priority' = priority \oplus \{(target?, newpri?)\}$

CreateTaskS_T

$\Delta Task$
$target? : TASK$
$newpri? : \mathbb{N}$

$state(target?) = nonexistent$
$newpri? > priority(running\_task)$
$\exists\, st? : STATE;\ tasks? : \mathbb{P}\, TASK;\ pri? : TASK \to \mathbb{N}$
$\mid st? = ready \land tasks? = tasks \cup \{(target?)\}$
$\land\ pri? = priority \oplus \{(target?, newpri?)\} \bullet Reschedule$

DeleteTaskN_T

$\Delta Task$
$target? : TASK$
$topReady! : TASK$

$target? \in tasks \setminus \{idle\}$
$state(target?) \in \{ready, blocked, suspended\}$
$tasks' = tasks \setminus \{target?\}$
$running\_task' = running\_task$
$state' = state \oplus \{(target?, nonexistent)\}$
$phys\_context' = phys\_context$
$log\_context' = log\_context \oplus \{(target?, bare\_context)\}$
$\Xi PrioData$
$topReady! = running\_task$

DeleteTaskS_T

$\Delta Task$
$target? : TASK$
$topReady! : TASK$

$target? \in tasks \setminus \{idle\}$
$state(target?) \in \{running\}$
$state(topReady!) = ready$
$\forall\, t : state^{\sim}(\!|\ \{ready\}\ |\!) \bullet priority(topReady!) \geq priority(t)$
$tasks' = tasks \setminus \{target?\}$
$running\_task' = topReady!$
$state' = state \oplus \{(topReady!, running), (target?, nonexistent)\}$
$phys\_context' = log\_context(topReady!)$
$log\_context' = log\_context \oplus \{(target?, bare\_context)\}$
$\Xi PrioData$

SuspendTaskN_T

$\Delta Task$
$target? : TASK$
$topReady! : TASK$

$target? \in tasks \setminus \{idle\}$
$state(target?) \in \{ready, blocked\}$
$\Xi TaskData$
$state' = state \oplus \{(target?, suspended)\}$
$\Xi ContextData$
$\Xi PrioData$
$topReady! = running\_task$

SuspendTaskS_T

$\Delta Task$
$target? : TASK$
$topReady! : TASK$

$target? \in tasks \setminus \{idle\}$
$state(target?) \in \{running\}$
$state(topReady!) = ready$
$\forall t : state^{\sim}(\!| \{ready\} |\!) \bullet priority(topReady!) \geq priority(t)$
$\exists st? : STATE \mid st? = suspended$
$\quad \bullet Reschedule[tasks/tasks?, priority/pri?, topReady!/target?]$

SuspendTaskO_T

$\Xi Task$
$target? : TASK$
$topReady! : TASK$

$state(target?) \in \{suspended\}$
$topReady! = running\_task$

ResumeTaskN_T

$\Delta Task$
$target? : TASK$

$state(target?) = suspended$
$priority(target?) \leq priority(running\_task)$
$\Xi TaskData$
$state' = state \oplus \{(target?, ready)\}$
$\Xi ContextData$
$\Xi PrioData$

ResumeTaskS_T

$\Delta Task$
$target? : TASK$

$state(target?) = suspended$
$priority(target?) > priority(running\_task)$
$\exists\, st? : STATE \mid st? = ready \bullet Reschedule[tasks/tasks?, priority/pri?]$

ChangeTaskPriorityN_T

$\Delta Task$
$target? : TASK$
$newpri? : \mathbb{N}$
$topReady! : TASK$

$state(target?) = ready \Rightarrow newpri? \leq priority(running\_task)$
$state(target?) = running \Rightarrow (\forall\, t : state^{\sim}(\!|\, \{ready\} \,|\!)$
$\quad \bullet\ newpri? \geq priority(t))$
$state(target?) \neq nonexistent$
$target? = idle \Rightarrow newpri? = 0$
$\Xi TaskData$
$\Xi StateData$
$\Xi ContextData$
$priority' = priority \oplus \{(target?, newpri?)\}$
$topReady! = running\_task$

ChangeTaskPriorityS_T

$\Delta Task$
$target? : TASK$
$newpri? : \mathbb{N}$
$topReady! : TASK$

$state(target?) = ready$
$newpri? > priority(running\_task)$
$target? = idle \Rightarrow newpri? = 0$
$\exists\, st? : STATE;\ pri? : TASK \rightarrow \mathbb{N}$
$\quad \mid st? = ready$
$\quad\quad \wedge\ pri? = priority \oplus \{(target?, newpri?)\}$
$\quad \bullet\ Reschedule[tasks/tasks?]$
$topReady! = target?$

ChangeTaskPriorityD_T

$\Delta Task$
$target? : TASK$
$newpri? : \mathbb{N}$

$$
\begin{array}{|l}
topReady! : TASK \\
\hline
state(target?) = running \\
target? = idle \Rightarrow newpri? = 0 \\
state(topReady!) = ready \\
\forall\, t : state^{\sim}(\!|\, \{ready\} \,|\!) \bullet priority(topReady!) \geq priority(t) \\
newpri? < priority(topReady!) \\
\exists\, st? : STATE;\ pri? : TASK \to \mathbb{N} \\
\quad | \ st? = ready \\
\qquad \wedge\ pri? = priority \oplus \{(target?, newpri?)\} \\
\quad \bullet\ Reschedule[tasks/tasks?, topReady!/target?]
\end{array}
$$

Multi_Task

$$
\begin{array}{|l}
subTask : cores \to Task \\
exeCore : TASK \twoheadrightarrow cores \\
\hline
\forall\, c1, c2 : cores \mid c1 \neq c2 \bullet \\
(subTask\ c1).tasks \cap (subTask\ c2).tasks = \emptyset \\
\mathrm{dom}\ exeCore \in \mathbb{F}\ TASK \\
\mathrm{dom}\ exeCore = \bigcup \{ c : cores \bullet (subTask(c)).tasks \}
\end{array}
$$

Init

$$
\begin{array}{|l}
Multi\_Task' \\
\hline
\mathrm{dom}\ exeCore' = idles \\
\forall\, c : cores \bullet \exists\, Task' \\
\quad | \ Init\_Task \bullet subTask'(c) = \theta Task' \\
\qquad \wedge\ exeCore'((subTask'(c)).idle) = c
\end{array}
$$

PromoteC

$$
\begin{array}{|l}
\Delta Multi\_Task \\
\Delta Task \\
target? : TASK \\
executeCore : cores \\
\hline
target? \notin \mathrm{dom}\ exeCore \\
subTask(executeCore) = \theta Task \\
subTask' = subTask \oplus \{(executeCore, \theta Task')\} \\
exeCore' = exeCore \oplus \{(target?, executeCore)\}
\end{array}
$$

$$createTaskN\_MT \mathrel{\widehat=} \exists\, \Delta Task \bullet CreateTaskN\_T \wedge PromoteC$$

$$createTaskS\_MT \mathrel{\widehat=} \exists\, \Delta Task \bullet CreateTaskS\_T \wedge PromoteC$$

findACore_MT

$$
\begin{array}{l}
Multi\_Task \\
target? : TASK \\
newpri? : \mathbb{N} \\
executeCore? : CORE \\
executeCore : CORE \\
\hline
executeCore? \notin cores \\
executeCore \in cores \\
\exists\, tcs, cs : \mathbb{F}\, cores \\
\quad \mid tcs = \{\, pc : cores;\ subS : Task \\
\qquad \mid subS = subTask(pc) \\
\qquad\quad \land\ newpri? > subS.priority(subS.running\_task) \bullet pc \,\} \\
\quad \bullet\ (tcs = \emptyset \Rightarrow cs = cores) \\
\qquad \land\ (tcs \neq \emptyset \Rightarrow cs = tcs) \\
\qquad \land\ (\forall\, oc : cs \bullet executeCore \in cs \\
\qquad\quad \land\ \#(exeCore^{\sim}(\!|\ \{executeCore\}\ |\!)) \leq \\
\qquad\qquad \#(exeCore^{\sim}(\!|\ \{oc\}\ |\!)))
\end{array}
$$

$$
CreateTaskN\_MT \mathrel{\widehat=} ([executeCore?, executeCore : CORE \mid executeCore? \in cores \land executeCore = executeCore?] \lor findACore\_MT) \land createTaskN\_MT
$$

$$
CreateTaskS\_MT \mathrel{\widehat=} ([executeCore?, executeCore : CORE \mid executeCore? \in cores \land executeCore = executeCore?] \lor findACore\_MT) \land createTaskS\_MT
$$

PromoteD

$$
\begin{array}{l}
\Delta Multi\_Task \\
\Delta Task \\
target? : TASK \\
\hline
target? \in \operatorname{dom} exeCore \\
subTask(exeCore(target?)) = \theta Task \\
subTask' = subTask \oplus \{(exeCore(target?), \theta Task')\} \\
exeCore' = \{target?\} \lhd exeCore
\end{array}
$$

$$
DeleteTaskN\_MT \mathrel{\widehat=} \exists\, \Delta Task \bullet DeleteTaskN\_T \land PromoteD
$$

$$
DeleteTaskS\_MT \mathrel{\widehat=} \exists\, \Delta Task \bullet DeleteTaskS\_T \land PromoteD
$$

Promote

$\Delta Multi\_Task$
$\Delta Task$
$target? : TASK$

$target? \in \mathrm{dom}\ exeCore$
$subTask(exeCore(target?)) = \theta Task$
$subTask' = subTask \oplus \{(exeCore(target?), \theta Task')\}$
$exeCore' = exeCore$

$SuspendTaskN\_MT \mathrel{\widehat=} \exists \Delta Task \bullet SuspendTaskN\_T \land Promote$

$SuspendTaskS\_MT \mathrel{\widehat=} \exists \Delta Task \bullet SuspendTaskS\_T \land Promote$

$SuspendTaskO\_MT \mathrel{\widehat=} \exists \Delta Task \bullet SuspendTaskO\_T \land Promote$

$ResumeTaskN\_MT \mathrel{\widehat=} \exists \Delta Task \bullet ResumeTaskN\_T \land Promote$

$ResumeTaskS\_MT \mathrel{\widehat=} \exists \Delta Task \bullet ResumeTaskS\_T \land Promote$

$ChangeTaskPriorityN\_MT \mathrel{\widehat=}$
$\quad \exists \Delta Task \bullet ChangeTaskPriorityN\_T \land Promote$

$ChangeTaskPriorityS\_MT \mathrel{\widehat=}$
$\quad \exists \Delta Task \bullet ChangeTaskPriorityS\_T \land Promote$

$ChangeTaskPriorityD\_MT \mathrel{\widehat=}$
$\quad \exists \Delta Task \bullet ChangeTaskPriorityD\_T \land Promote$

$MigrationN\_MT$

$\Delta Multi\_Task$
$target? : TASK$
$topReady! : TASK$
$newCore? : cores$
$srcSys, tarSys : Task$

$target? \in \mathrm{dom}\ exeCore$
$srcSys = subTask(exeCore(target?))$
$tarSys = subTask(newCore?)$
$srcSys.state(target?) \in \{ready, blocked, suspended\}$
$srcSys.state(target?) = ready \Rightarrow$
$\quad srcSys.priority(target?) \leq tarSys.priority(tarSys.running\_task)$

$target? \notin idles$
$newCore? \neq exeCore(target?)$
$\{exeCore(target?), newCore?\} \lhd subTask' =$
$\quad \{exeCore(target?), newCore?\} \lhd subTask$
$\exists \Delta Task \bullet$
$\quad subTask(exeCore(target?)) = \theta Task$
$\quad \land\ DeleteTaskN\_T$
$\quad \land\ subTask'(exeCore(target?)) = \theta Task'$
$\exists \Delta Task;\ tpri : \mathbb{N} \mid$
$\quad tpri = (srcSys.priority(target?)) \bullet$
$\quad\quad subTask(newCore?) = \theta Task$
$\quad\quad \land\ CreateTaskN\_T[tpri/newpri?]$
$\quad\quad \land\ subTask'(newCore?) = \theta Task'$
$exeCore' = exeCore \oplus \{(target?, newCore?)\}$

MigrationS_MT

$\Delta Multi\_Task$
$target? : TASK$
$topReady! : TASK$
$newCore? : cores$
$srcSys, tarSys : Task$

$target? \in \mathrm{dom}\ exeCore$
$srcSys = subTask(exeCore(target?))$
$tarSys = subTask(newCore?)$
$srcSys.state(target?) = ready$
$srcSys.priority(target?) > tarSys.priority(tarSys.running\_task)$
$target? \notin idles$
$newCore? \neq exeCore(target?)$
$\{exeCore(target?), newCore?\} \lhd subTask' =$
$\quad \{exeCore(target?), newCore?\} \lhd subTask$
$\exists \Delta Task \bullet$
$subTask(exeCore(target?)) = \theta Task$
$\land\ DeleteTaskN\_T$
$\land\ subTask'(exeCore(target?)) = \theta Task'$
$\exists \Delta Task;\ tpri : \mathbb{N} \mid$
$tpri = (srcSys.priority(target?)) \bullet$
$subTask(newCore?) = \theta Task$
$\land\ CreateTaskS\_T[tpri/newpri?]$
$\land\ subTask'(newCore?) = \theta Task'$
$exeCore' = exeCore \oplus \{(target?, newCore?)\}$

MigrationRuN_MT

$\Delta Multi\_Task$

$target? : TASK$
$topReady! : TASK$
$newCore? : cores$
$srcSys, tarSys : Task$

$target? \in \mathrm{dom}\ exeCore$
$srcSys = subTask(exeCore(target?))$
$tarSys = subTask(newCore?)$
$srcSys.state(target?) = running$
$srcSys.priority(target?) \leq tarSys.priority(tarSys.running\_task)$
$target? \notin idles$
$newCore? \neq exeCore(target?)$
$\{exeCore(target?), newCore?\} \lhd subTask' =$
$\quad \{exeCore(target?), newCore?\} \lhd subTask$
$\exists \Delta Task \bullet$
$subTask(exeCore(target?)) = \theta Task$
$\land\ DeleteTaskS\_T$
$\land\ subTask'(exeCore(target?)) = \theta Task'$
$\exists \Delta Task;\ tpri : \mathbb{N} \mid$
$tpri = (srcSys.priority(target?)) \bullet$
$subTask(newCore?) = \theta Task$
$\land\ CreateTaskN\_T[tpri/newpri?]$
$\land\ subTask'(newCore?) = \theta Task'$
$exeCore' = exeCore \oplus \{(target?, newCore?)\}$

MigrationRuS_MT

$\Delta Multi\_Task$
$target? : TASK$
$topReady! : TASK$
$newCore? : cores$
$srcSys, tarSys : Task$

$target? \in \mathrm{dom}\ exeCore$
$srcSys = subTask(exeCore(target?))$
$tarSys = subTask(newCore?)$
$srcSys.state(target?) = running$
$srcSys.priority(target?) > tarSys.priority(tarSys.running\_task)$
$target? \notin idles$
$newCore? \neq exeCore(target?)$
$\{exeCore(target?), newCore?\} \lhd subTask' =$
$\quad \{exeCore(target?), newCore?\} \lhd subTask$
$\exists \Delta Task \bullet$
$subTask(exeCore(target?)) = \theta Task$
$\land\ DeleteTaskS\_T$
$\land\ subTask'(exeCore(target?)) = \theta Task'$
$\exists \Delta Task;\ tpri : \mathbb{N} \mid$

$tpri = (srcSys.priority(target?)) \bullet$
$subTask(newCore?) = \theta Task$
$\land\ CreateTaskS\_T[tpri/newpri?]$
$\land\ subTask'(newCore?) = \theta Task'$
$exeCore' = exeCore \oplus \{(target?, newCore?)\}$

$CN\_T \mathrel{\widehat=} CreateTaskN\_T$

$CS\_T \mathrel{\widehat=} CreateTaskS\_T$

$DN\_T \mathrel{\widehat=} DeleteTaskN\_T$

$DS\_T \mathrel{\widehat=} DeleteTaskS\_T$

$SN\_T \mathrel{\widehat=} SuspendTaskN\_T$

$SS\_T \mathrel{\widehat=} SuspendTaskS\_T$

$SO\_T \mathrel{\widehat=} SuspendTaskO\_T$

$RN\_T \mathrel{\widehat=} ResumeTaskN\_T$

$RS\_T \mathrel{\widehat=} ResumeTaskS\_T$

$ChN\_T \mathrel{\widehat=} ChangeTaskPriorityN\_T$

$ChS\_T \mathrel{\widehat=} ChangeTaskPriorityS\_T$

$ChD\_T \mathrel{\widehat=} ChangeTaskPriorityD\_T$

getRunningTask

$\Xi Multi\_Task$
$core? : cores$
$RT! : TASK$

$RT! = (subTask(core?)).running\_task$

getPriority

$\Xi Multi\_Task$
$task? : TASK$
$PRIORITY! : \mathbb{N}$

$\exists\, core : cores \mid task? \in (subTask(core)).tasks$
$\bullet\ PRIORITY! = (subTask(core)).priority(task?)$

Appendix M

VCC Annotated Source Code

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #include "FreeRTOS.h"
6  #include "task.h"
7
8  _(dynamic_owns) typedef struct tskTaskControlBlock
9  {
10     xListItem               xGenericListItem;       /*< The list that the state list item
           of a task is reference from denotes the state of that task (Ready, Blocked,
           Suspended ). */
11     xListItem               xEventListItem;     /*< Used to reference a task from an event
            list. */
12     unsigned portBASE_TYPE  uxPriority;         /*< The priority of the task.  0 is the
           lowest priority. */
13
14         _(invariant uxPriority < configMAX_PRIORITIES)
15         _(invariant \mine(&xGenericListItem))
16         _(invariant \mine(&xEventListItem))
17 } tskTCB;
18
19 PRIVILEGED_DATA tskTCB * volatile pxCurrentTCB = NULL;
20
21 PRIVILEGED_DATA static xList pxReadyTasksLists[ configMAX_PRIORITIES ]; /*< Prioritised
       ready tasks. */
22 PRIVILEGED_DATA static xList xDelayedTaskList1;                         /*< Delayed tasks.
       */
23 PRIVILEGED_DATA static xList xDelayedTaskList2;                         /*< Delayed tasks
       (two lists are used - one for delays that have overflowed the current tick count. */
24 PRIVILEGED_DATA static xList * volatile pxDelayedTaskList ;            /*< Points to the
       delayed task list currently being used. */
25 PRIVILEGED_DATA static xList * volatile pxOverflowDelayedTaskList;     /*< Points to the
       delayed task list currently being used to hold tasks that have overflowed the current
       tick count. */
26 PRIVILEGED_DATA static xList xPendingReadyList;                        /*< Tasks that
       have been readied while the scheduler was suspended.  They will be moved to the ready
       queue when the scheduler is resumed. */
27
28 PRIVILEGED_DATA static xList xTasksWaitingTermination;              /*< Tasks that have
       been deleted - but the their memory not yet freed. */
29
30 PRIVILEGED_DATA static xList xSuspendedTaskList;
31
32 _(def \bool excList(){
33
34     \bool tmp = (
35         \forall int i1, i2; ( 0 <= i1 && i1 < i2 && i2 < configMAX_PRIORITIES )
```

```
36              ==> !((&pxReadyTasksLists[i1]) == (&pxReadyTasksLists[i2]))
37      ) && (
38          \forall int i; ( 0 <= i && i < configMAX_PRIORITIES )
39              ==> (!((&pxReadyTasksLists[i]) == &xDelayedTaskList1)
40                && !((&pxReadyTasksLists[i]) == &xDelayedTaskList2)
41                && !((&pxReadyTasksLists[i]) == &xSuspendedTaskList)
42                && !((&pxReadyTasksLists[i]) == &xTasksWaitingTermination))
43      ) && (
44          &xDelayedTaskList1 != &xDelayedTaskList2
45      ) && (
46          &xDelayedTaskList1 != &xSuspendedTaskList
47      ) && (
48          &xDelayedTaskList1 != &xTasksWaitingTermination
49      ) && (
50          &xDelayedTaskList2 != &xSuspendedTaskList
51      ) && (
52          &xDelayedTaskList2 != &xTasksWaitingTermination
53      ) && (
54          &xSuspendedTaskList != &xTasksWaitingTermination
55      );
56
57      return tmp;
58  })
59
60  PRIVILEGED_DATA static xTaskHandle xIdleTaskHandle = NULL;          /*< Holds the handle
        of the idle task.  The idle task is created automatically when the scheduler is
        started. */
61
62  PRIVILEGED_DATA static volatile signed portBASE_TYPE xSchedulerRunning          = pdFALSE;
63
64  #define prvGetTCBFromHandle( pxHandle ) ( ( ( pxHandle ) == NULL ) ? ( tskTCB * )
        pxCurrentTCB : ( tskTCB * ) ( pxHandle ) )
65
66  _(ghost typedef tskTCB * TASK;)
67
68  _(ghost _(dynamic_owns) typedef struct{
69      \bool tasks[TASK];
70      STATE state[TASK];
71      //CONTEXT phys_context;
72      //CONTEXT log_context[TASK];
73
74      \natural priority[TASK];
75
76      //READY
77      _(invariant \forall TASK t; (tasks[t] && xSchedulerRunning != pdFALSE) ==>
78          ((((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &pxReadyTasksLists[((
                tskTCB *) t)->uxPriority] &&
79           ((tskTCB *) t) != pxCurrentTCB) <==> state[t] == ready))
80      //BLOCKED
81      _(invariant \forall TASK t; (tasks[t] && xSchedulerRunning != pdFALSE) ==>
82          ((((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &xDelayedTaskList1 ||
83            ((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &xDelayedTaskList2)
                <==> state[t] == blocked))
84      //SUSPENDED
85      _(invariant \forall TASK t; (tasks[t] && xSchedulerRunning != pdFALSE) ==>
86          (((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &xSuspendedTaskList
                <==> state[t] == suspended))
87      //RUNNING
88      _(invariant \forall TASK t; (tasks[t] && xSchedulerRunning != pdFALSE) ==>
89          (t == (TASK) pxCurrentTCB <==> state[t] == running))
90      _(invariant ((xList *) pxCurrentTCB->xGenericListItem.pvContainer) == &
            pxReadyTasksLists[pxCurrentTCB->uxPriority])
91      //NONEXISTENT
92      _(invariant \forall TASK t; (t->\closed && xSchedulerRunning != pdFALSE) ==>
93          ((((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                xTasksWaitingTermination ||
94            t == NULL) <==> state[t] == nonexistent))
95
96      _(invariant \forall TASK t; xSchedulerRunning != pdFALSE ==> (state[t] != nonexistent
            <==> tasks[t]))
97      _(invariant \forall TASK t; tasks[t] ==> \mine(t))
98      _(invariant \forall TASK t; tasks[t] ==> t->\closed)
99      _(invariant \forall TASK t; tasks[t] ==> priority[t] == t->uxPriority)
100     _(invariant \forall TASK t; tasks[t] ==> state[t] <= 4)
101     _(invariant \mine(\embedding(& xIdleTaskHandle)))
```

```
102     _(invariant \mine(\embedding(& pxCurrentTCB)))
103     _(invariant xSchedulerRunning != pdFALSE ==> (tasks[xIdleTaskHandle] && tasks[
            pxCurrentTCB]))
104     _(invariant xSchedulerRunning != pdFALSE && xIdleTaskHandle != NULL ==> priority[
            xIdleTaskHandle] == 0)
105     _(invariant \forall TASK t; xSchedulerRunning != pdFALSE && state[t] == ready ==>
            priority[pxCurrentTCB] >= priority[t])
106 } * FRTOS;)
107
108 //signed portBASE_TYPE xTaskGenericCreate(unsigned portBASE_TYPE uxPriority, xTaskHandle *
        pxCreatedTask _(ghost FRTOS FreeRTOS) _(ghost TASK *newTask))
109 signed portBASE_TYPE xTaskGenericCreate( pdTASK_CODE pxTaskCode, const signed char * const
        pcName, unsigned short usStackDepth, void *pvParameters, unsigned portBASE_TYPE
        uxPriority, xTaskHandle *pxCreatedTask, portSTACK_TYPE *puxStackBuffer, const
        xMemoryRegion * const xRegions _(ghost FRTOS FreeRTOS) _(ghost TASK *newTask) )
110     _(updates FreeRTOS)
111     _(requires \mutable(&xSchedulerRunning))
112     _(requires xSchedulerRunning == pdTRUE)
113     _(requires excList())
114
115     //_(writes newTask, pxCreatedTask,
116 //        &pxReadyTasksLists[uxPriority >= configMAX_PRIORITIES ? configMAX_PRIORITIES - (
            unsigned portBASE_TYPE ) 1U : uxPriority],
117 //        \embedding(&pxReadyTasksLists[uxPriority >= configMAX_PRIORITIES ?
            configMAX_PRIORITIES - ( unsigned portBASE_TYPE ) 1U : uxPriority]))
118     _(writes newTask, pxCreatedTask)
119
120     _(ensures \result == pdPASS ==> uxPriority > \old(pxCurrentTCB)->uxPriority ==> (
            FreeRTOS->state[(TASK)\old(pxCurrentTCB)]) == ready && (FreeRTOS->state[(TASK)
            pxCurrentTCB]) == running)
121     _(ensures \result == pdPASS ==> uxPriority <= \old(pxCurrentTCB)->uxPriority ==> (
            FreeRTOS->state[*newTask]) == ready)
122     _(ensures \result == pdPASS ==> FreeRTOS->priority[*newTask] ==
123         (\natural)(uxPriority < configMAX_PRIORITIES ? uxPriority : configMAX_PRIORITIES -
                ( unsigned portBASE_TYPE ) 1U))
124     _(ensures \result == pdPASS ==> pxCurrentTCB->\closed)
125     _(ensures \result == pdPASS ==> \fresh(*newTask))
126     _(ensures \forall TASK t; (FreeRTOS->tasks[t] && \old(FreeRTOS->state[t]) != FreeRTOS
            ->state[t]) ==> transition[\old(FreeRTOS->state[t])][FreeRTOS->state[t]])
127 {
128     _(assert \wrapped(FreeRTOS))
129     _(assert \inv(FreeRTOS))
130     _(assert xSchedulerRunning == pdTRUE)
131     _(assert FreeRTOS->tasks[pxCurrentTCB])
132     _(assert pxCurrentTCB \in FreeRTOS->\owns)
133
134     //_(assert \forall TASK t; FreeRTOS->tasks[t] ==> (t \in FreeRTOS->\owns && t->\closed
            ))
135     //_(assert \forall TASK t; FreeRTOS->tasks[t] ==> t->\closed)
136     _(assert \forall TASK t; FreeRTOS->tasks[t] ==> \inv(t))
137     //_(assert \forall TASK t; FreeRTOS->tasks[t] ==> FreeRTOS->priority[t] == t->
            uxPriority)
138     _(assert \forall TASK t; FreeRTOS->tasks[t] ==> FreeRTOS->priority[t] <
            configMAX_PRIORITIES)
139
140     _(assert \forall TASK t; FreeRTOS->tasks[t] ==> FreeRTOS->state[t] <= 4)
141     _(assert pxCurrentTCB->\closed)
142
143     signed portBASE_TYPE xReturn;
144     tskTCB * pxNewTCB;
145
146     pxNewTCB = (tskTCB *) malloc(sizeof(tskTCB));
147
148         _(assert pxNewTCB != pxCurrentTCB)
149         //_(assert \wrapped(pxCurrentTCB))
150
151         if ( pxNewTCB != NULL )
152         {
153             _(assert pxNewTCB)
154             _(assert \fresh(pxNewTCB))
155
156             /* Setup the newly allocated TCB with the initial state of the task. */
157             //prvInitialiseTCBVariables( pxNewTCB, pcName, uxPriority, xRegions,
                    usStackDepth );
158
```

```
159             pxNewTCB->uxPriority = (uxPriority < configMAX_PRIORITIES ? uxPriority :
                    configMAX_PRIORITIES - ( unsigned portBASE_TYPE ) 1U);
160             pxNewTCB->xGenericListItem.pvContainer = &pxReadyTasksLists[pxNewTCB->
                    uxPriority];
161
162             _(assert \writable(&(pxNewTCB->xGenericListItem)))
163             _(assert \writable(&(pxNewTCB->xEventListItem)))
164             _(wrap &(pxNewTCB->xGenericListItem))
165             _(wrap &(pxNewTCB->xEventListItem))
166
167             _(ghost pxNewTCB->\owns = (\objset) {&(pxNewTCB->xGenericListItem)})
168             _(ghost pxNewTCB->\owns += &(pxNewTCB->xEventListItem))
169
170             _(assert !(FreeRTOS \in pxNewTCB->\owns))
171             _(wrap pxNewTCB)
172
173             if ( ( void * ) pxCreatedTask != NULL )
174             {
175                 *pxCreatedTask = ( xTaskHandle ) pxNewTCB;
176             }
177
178             //_(assert \false)
179             taskENTER_CRITICAL();
180             {
181                 //...
182                 xReturn = pdPASS;
183             }
184             taskEXIT_CRITICAL();
185         }
186         else
187         {
188             _(assert \inv(FreeRTOS))
189             xReturn = errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY;
190         }
191
192         if( xReturn == pdPASS )
193         {
194             _(unwrapping FreeRTOS){
195                 _(assert \inv(FreeRTOS))
196
197                 _(ghost {
198                     FreeRTOS->priority[(TASK) pxNewTCB] = pxNewTCB->uxPriority;
199                     FreeRTOS->state[(TASK) pxNewTCB] = ready;
200                     FreeRTOS->tasks[(TASK) pxNewTCB] = \true;
201                     FreeRTOS->\owns += pxNewTCB;
202                     *newTask = pxNewTCB;
203                 })
204
205                 _(assert FreeRTOS->state[(TASK) pxNewTCB] == ready)
206                 _(assert FreeRTOS->state[(TASK) pxCurrentTCB] == running)
207                 _(assert FreeRTOS->priority[(TASK) pxNewTCB] == pxNewTCB->uxPriority)
208                 _(assert \forall TASK t; (FreeRTOS->tasks[t] && ((tskTCB *)t != pxNewTCB)
                        && xSchedulerRunning != pdFALSE) ==>
209                     (
210                         (((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                                pxReadyTasksLists[((tskTCB *) t)->uxPriority] &&
211                         ((tskTCB *) t) != pxCurrentTCB) <==> FreeRTOS->state[t] == ready
212                     )
213                 )
214
215                 _(assert \forall TASK t; (FreeRTOS->tasks[t] && ((tskTCB *)t != pxNewTCB)
                        && xSchedulerRunning != pdFALSE) ==>
216                     (
217                         (((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                                xDelayedTaskList1 ||
218                          ((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                                 xDelayedTaskList2)
219                             <==> FreeRTOS->state[t] == blocked
220                     )
221                 )
222
223                 _(assert \forall TASK t; (FreeRTOS->tasks[t] && ((tskTCB *)t != pxNewTCB)
                        && xSchedulerRunning != pdFALSE) ==>
224                     (
```

```
225                         ((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                                xSuspendedTaskList
226                         <==> FreeRTOS->state[t] == suspended
227                     )
228                 )
229
230                 _(assert \forall TASK t; (t->\closed && ((tskTCB *)t != pxNewTCB) &&
                        xSchedulerRunning != pdFALSE) ==>
231                     (
232                         (((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                                xTasksWaitingTermination ||
233                           t == NULL)
234                         <==> FreeRTOS->state[t] == nonexistent
235                     )
236                 )
237
238
239                 if( _(atomic_read \embedding(&pxCurrentTCB))
240                     pxCurrentTCB->uxPriority < uxPriority )
241                 {
242                     _(ghost {
243                         FreeRTOS->state[(TASK) pxCurrentTCB] = ready;
244                         FreeRTOS->state[(TASK) pxNewTCB] = running;
245                     })
246
247                     _(assert FreeRTOS->state[(TASK) \old(pxCurrentTCB)] == ready)
248
249                     //portYIELD_WITHIN_API();
250                     _(atomic \embedding(&pxCurrentTCB)){
251                         pxCurrentTCB = pxNewTCB;
252                         _(bump_volatile_version \embedding(&pxCurrentTCB))
253                     }
254                 }
255
256                 _(assert \old(pxCurrentTCB)->uxPriority >= uxPriority ==> (FreeRTOS->state
                        [(TASK) pxNewTCB] == ready && FreeRTOS->state[(TASK) pxCurrentTCB] ==
                        running))
257                 _(assert \old(pxCurrentTCB)->uxPriority < uxPriority ==>
258                     (FreeRTOS->state[(TASK) pxNewTCB] == running &&
259                         pxNewTCB == pxCurrentTCB &&
260                         FreeRTOS->state[(TASK) \old(pxCurrentTCB)] == ready
261                     )
262                 )
263                 _(assert FreeRTOS->priority[(TASK) pxNewTCB] == pxNewTCB->uxPriority)
264                 _(assert FreeRTOS->priority[(TASK) \old(pxCurrentTCB)] == \old(
                        pxCurrentTCB)->uxPriority)
265                 _(assert FreeRTOS->tasks[(TASK) \old(pxCurrentTCB)])
266
267                 _(assume \forall TASK t; (FreeRTOS->tasks[t] && ((tskTCB *)t != pxNewTCB)
                        && xSchedulerRunning != pdFALSE) ==>
268                     (
269                         (((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                                pxReadyTasksLists[((tskTCB *) t)->uxPriority] &&
270                         ((tskTCB *) t) != pxCurrentTCB)
271                         <==> FreeRTOS->state[t] == ready
272                     )
273                 )
274
275                 _(assume \forall TASK t; (FreeRTOS->tasks[t] && ((tskTCB *)t != pxNewTCB)
                        && xSchedulerRunning != pdFALSE) ==>
276                     (
277                         (((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                                xDelayedTaskList1 ||
278                          ((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                                xDelayedTaskList2)
279                         <==> FreeRTOS->state[t] == blocked
280                     )
281                 )
282
283                 _(assume \forall TASK t; (FreeRTOS->tasks[t] && ((tskTCB *)t != pxNewTCB)
                        && xSchedulerRunning != pdFALSE) ==>
284                     (
285                         ((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                                xSuspendedTaskList
286                         <==> FreeRTOS->state[t] == suspended
```

```
287                     )
288                 )
289
290                 _(assume \forall TASK t; (t->\closed && ((tskTCB *)t != pxNewTCB) &&
                        xSchedulerRunning != pdFALSE) ==>
291                     (
292                         (((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                                xTasksWaitingTermination ||
293                           t == NULL)
294                             <==> FreeRTOS->state[t] == nonexistent
295                     )
296                 )
297
298                 _(assert \forall TASK t; FreeRTOS->tasks[t] ==> t->\closed)
299
300
301                 //_(assume \inv(FreeRTOS))
302             }//wrapping FreeRTOS
303
304         }
305     _(assume Trans())
306     return xReturn;
307 }
308
309 void vTaskDelete( xTaskHandle pxTaskToDelete _(ghost FRTOS FreeRTOS) _(ghost TASK topReady
        ))
310     _(updates FreeRTOS)
311
312     _(requires FreeRTOS->tasks[topReady])
313     _(requires FreeRTOS->state[topReady] == ready)
314     _(requires \forall TASK rts;
315         (FreeRTOS->tasks[rts] && FreeRTOS->state[rts] == ready)
316             ==> FreeRTOS->priority[topReady] >= FreeRTOS->priority[rts])
317     _(requires topReady != (TASK) pxTaskToDelete)
318     _(requires topReady != (TASK) pxCurrentTCB)
319
320     _(requires FreeRTOS->tasks[(TASK) pxTaskToDelete])
321     _(requires (tskTCB *)pxTaskToDelete != (tskTCB *)xIdleTaskHandle)
322     //_(requires (TASK) pxTaskToDelete->\closed)
323
324     _(requires \mutable(&xSchedulerRunning))
325     _(requires xSchedulerRunning == pdTRUE)
326     _(requires excList())
327
328     _(ensures ((TASK) pxTaskToDelete)->\closed)
329     _(ensures pxTaskToDelete != NULL ==> ! FreeRTOS->tasks[(TASK) pxTaskToDelete])
330     _(ensures pxTaskToDelete == NULL ==> ! FreeRTOS->tasks[(TASK) \old(pxCurrentTCB)])
331     _(ensures pxTaskToDelete == NULL ==> (TASK) pxCurrentTCB == topReady)
332     _(ensures \forall TASK t; (FreeRTOS->tasks[t] && \old(FreeRTOS->state[t]) != FreeRTOS
            ->state[t]) ==> transition[\old(FreeRTOS->state[t])][FreeRTOS->state[t]])
333 {
334     _(assert \wrapped(FreeRTOS))
335     _(assert \inv(FreeRTOS))
336     _(assert xSchedulerRunning == pdTRUE)
337     //_(assert FreeRTOS->tasks[(TASK) pxTaskToDelete])
338     //_(assert pxCurrentTCB \in FreeRTOS->\owns)
339     _(assert \forall TASK t; FreeRTOS->tasks[t] ==> \inv(t))
340     _(assert \forall TASK t; FreeRTOS->tasks[t] ==> FreeRTOS->priority[t] <
            configMAX_PRIORITIES)
341     _(assert \forall TASK t; FreeRTOS->tasks[t] ==> FreeRTOS->state[t] <= 4)
342     _(assert pxCurrentTCB->\closed)
343     _(assert ((TASK)pxTaskToDelete)->\closed)
344
345     tskTCB *pxTCB;
346
347     _(unwrapping FreeRTOS){
348         taskENTER_CRITICAL();
349         {
350             _(assert \inv(FreeRTOS))
351             ///* Ensure a yield is performed if the current task is being
352             //deleted. */
353             //
354
355             _(atomic \embedding(&pxCurrentTCB)){
356                 if ( pxTaskToDelete == pxCurrentTCB )
```

```
357                 {
358                     pxTaskToDelete = NULL;
359                 }
360             }
361             //
362             ///* If null is passed in here then we are deleting ourselves. */
363             //pxTCB = prvGetTCBFromHandle( pxTaskToDelete );
364             _(atomic \embedding(&pxCurrentTCB)){
365                 pxTCB = pxTaskToDelete != NULL ? (tskTCB *) pxTaskToDelete
366                                                : pxCurrentTCB;
367             }
368             _(assert (TASK) pxTaskToDelete == (TASK) pxCurrentTCB ==>
369                     pxTCB == pxCurrentTCB)
370             _(assert \wrapped(pxTCB))
371 
372             _(unwrapping pxTCB){
373                 /* Remove task from the ready list and place in the termination list.
374                 This will stop the task from be scheduled.  The idle task will check
375                 the termination list and free up any memory allocated by the
376                 scheduler for the TCB and stack. */
377 
378                 _(assert \wrapped(&(pxTCB->xGenericListItem)))
379                 if( uxListRemove( ( xListItem * ) &( pxTCB->xGenericListItem ) ) == 0 )
380                 {
381                     //taskRESET_READY_PRIORITY( pxTCB->uxPriority );
382                 }
383 
384                 //_(assert \wrapped(&(pxTCB->xGenericListItem)))
385                 //_(unwrapping &(pxTCB->xGenericListItem)) {
386                 //    pxTCB->xGenericListItem.pvContainer = NULL;
387                 //}
388 
389                 _(assert pxTCB->xGenericListItem.pvContainer == NULL)
390 
391                 /* Is the task waiting on an event also? */
392                 if( pxTCB->xEventListItem.pvContainer != NULL )
393                 {
394                     uxListRemove( &( pxTCB->xEventListItem ) );
395                 }
396 
397                 _(assert pxTCB->xEventListItem.pvContainer == NULL)
398 
399                 vListInsertEnd( ( xList * ) &xTasksWaitingTermination, &( pxTCB->
                        xGenericListItem ) );
400                 _(assert (xList *)(pxTCB->xGenericListItem.pvContainer) == (xList *) &
                        xTasksWaitingTermination)
401             }
402 
403             _(ghost {
404                 FreeRTOS->state[(TASK) pxTCB] = nonexistent;
405                 FreeRTOS->tasks[(TASK) pxTCB] = \false;
406             })
407 
408         }
409         taskEXIT_CRITICAL();
410 
411         _(assert \forall TASK t; (FreeRTOS->tasks[t] && ((tskTCB *)t != pxTCB) &&
                xSchedulerRunning != pdFALSE) ==>
412             (
413                 (((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                        pxReadyTasksLists[((tskTCB *) t)->uxPriority] &&
414                 ((tskTCB *) t) != pxCurrentTCB)
415                     <==> FreeRTOS->state[t] == ready
416             )
417         )
418 
419         _(assert \forall TASK t; (FreeRTOS->tasks[t] && ((tskTCB *)t != pxTCB) &&
                xSchedulerRunning != pdFALSE) ==>
420             (
421                 (((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                        xDelayedTaskList1 ||
422                     ((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                            xDelayedTaskList2)
423                     <==> FreeRTOS->state[t] == blocked
424             )
```

```
425         )
426
427         _(assert \forall TASK t; (FreeRTOS->tasks[t] && ((tskTCB *)t != pxTCB) &&
                xSchedulerRunning != pdFALSE) ==>
428             (
429                 ((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                        xSuspendedTaskList
430                     <==> FreeRTOS->state[t] == suspended
431             )
432         )
433
434         //_(assert \forall TASK t; (t->\closed && ((tskTCB *)t != pxTCB) &&
                xSchedulerRunning != pdFALSE) ==>
435         //     (
436         //         (((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                xTasksWaitingTermination ||
437         //             t == NULL)
438         //             <==> FreeRTOS->state[t] == nonexistent
439         //     )
440         //)
441
442         /* Force a reschedule if we have just deleted the current task. */
443
444         if( xSchedulerRunning != pdFALSE )
445         {
446             if( ( void * ) pxTaskToDelete == NULL )
447             {
448                 //portYIELD_WITHIN_API();
449                 _(atomic \embedding(&pxCurrentTCB)){
450                     _(ghost pxCurrentTCB = topReady)
451                     _(bump_volatile_version \embedding(&pxCurrentTCB))
452                 }
453                 _(ghost FreeRTOS->state[topReady] = running)
454
455                 _(assert FreeRTOS->state[(TASK) pxTCB] == nonexistent)
456                 _(assert (xList *)(pxTCB->xGenericListItem.pvContainer) == &
                        xTasksWaitingTermination)
457                 _(assert !FreeRTOS->tasks[(TASK) pxTCB])
458
459                 _(assert FreeRTOS->state[(TASK) pxCurrentTCB] == running)
460                 _(assert \old(pxCurrentTCB) == pxTCB)
461                 _(assert FreeRTOS->state[(TASK) \old(pxCurrentTCB)] == nonexistent)
462             }
463         }
464
465         _(assert pxTaskToDelete != NULL ==> FreeRTOS->state[topReady] == ready)
466
467         _(assume \forall TASK t; (FreeRTOS->tasks[t] && ((tskTCB *)t != pxTCB) &&
                xSchedulerRunning != pdFALSE) ==>
468             (
469                 (((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                        pxReadyTasksLists[((tskTCB *) t)->uxPriority] &&
470                 ((tskTCB *) t) != pxCurrentTCB)
471                     <==> FreeRTOS->state[t] == ready
472             )
473         )
474
475         _(assume \forall TASK t; (FreeRTOS->tasks[t] && ((tskTCB *)t != pxTCB) &&
                xSchedulerRunning != pdFALSE) ==>
476             (
477                 (((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                        xDelayedTaskList1 ||
478                     ((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                            xDelayedTaskList2)
479                     <==> FreeRTOS->state[t] == blocked
480             )
481         )
482
483         _(assume \forall TASK t; (FreeRTOS->tasks[t] && ((tskTCB *)t != pxTCB) &&
                xSchedulerRunning != pdFALSE) ==>
484             (
485                 ((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                        xSuspendedTaskList
486                     <==> FreeRTOS->state[t] == suspended
487             )
```

```
488         )
489
490         _(assume \forall TASK t; (t->\closed && ((tskTCB *)t != pxTCB) &&
                xSchedulerRunning != pdFALSE) ==>
491             (
492                 (((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                        xTasksWaitingTermination ||
493                     t == NULL)
494                     <==> FreeRTOS->state[t] == nonexistent
495             )
496         )
497
498         _(assert \forall TASK t; FreeRTOS->tasks[t] ==> t->\closed)
499
500     }
501
502     _(assume Trans())
503     //_(assert \false)
504 }
505
506 #if ( INCLUDE_uxTaskPriorityGet == 1 )
507
508     unsigned portBASE_TYPE uxTaskPriorityGet( xTaskHandle pxTask _(ghost FRTOS FreeRTOS))
509         _(maintains \wrapped(FreeRTOS))
510         _(requires FreeRTOS->tasks[(TASK) pxTask])
511         _(requires xSchedulerRunning == pdTRUE)
512         _(requires excList())
513
514         _(writes FreeRTOS)
515
516         _(ensures \result == FreeRTOS->priority[(TASK) pxTask])
517     {
518     tskTCB *pxTCB;
519     unsigned portBASE_TYPE uxReturn;
520
521         taskENTER_CRITICAL();
522         {
523             /* If null is passed in here then we are changing the
524             priority of the calling function. */
525             //pxTCB = prvGetTCBFromHandle( pxTask );
526
527             _(unwrapping FreeRTOS){
528                 _(assert \inv(FreeRTOS))
529                 pxTCB = pxTask != NULL ? (tskTCB *) pxTask
530                                        : pxCurrentTCB;
531
532                 _(assert \wrapped(pxTCB))
533                 uxReturn = pxTCB->uxPriority;
534             }
535         }
536         taskEXIT_CRITICAL();
537
538         return uxReturn;
539     }
540
541 #endif
542
543 #if ( INCLUDE_vTaskPrioritySet == 1 )
544
545     void vTaskPrioritySet( xTaskHandle pxTask, unsigned portBASE_TYPE uxNewPriority _(
            ghost FRTOS FreeRTOS) _(ghost TASK topReady) )
546         _(updates FreeRTOS)
547         _(requires \mutable(&xSchedulerRunning))
548         _(requires xSchedulerRunning == pdTRUE)
549         _(requires excList())
550
551         _(requires FreeRTOS->tasks[topReady])
552         //_(requires FreeRTOS->state[topReady] == ready)
553         //_(requires \forall TASK rts;
554         //     (FreeRTOS->tasks[rts] && FreeRTOS->state[rts] == ready)
555         //         ==> FreeRTOS->priority[topReady] >= FreeRTOS->priority[rts])
556         //_(requires topReady != (TASK) pxTask)
557         //_(requires topReady != (TASK) pxCurrentTCB)
558
559         _(requires FreeRTOS->tasks[(TASK) pxTask])
```

```
560         _(requires (tskTCB *) pxTask == (tskTCB *) xIdleTaskHandle ==> uxNewPriority == 0)
561
562         _(ensures ((TASK) pxTask)->\closed)
563         _(ensures FreeRTOS->tasks[(TASK) pxTask])
564         _(ensures FreeRTOS->priority[(TASK) pxTask] ==
565             (\natural)(uxNewPriority < configMAX_PRIORITIES ? uxNewPriority :
                    configMAX_PRIORITIES - ( unsigned portBASE_TYPE ) 1U))
566         _(ensures (FreeRTOS->state[(TASK) pxTask] == ready && FreeRTOS->priority[(TASK)
                pxTask] > \old(pxCurrentTCB)->uxPriority) ==>
567             (FreeRTOS->state[(TASK) \old(pxCurrentTCB)] == ready && FreeRTOS->state[(TASK)
                     pxTask] == running)
568         )
569         _(ensures ((pxTask == NULL || (tskTCB *) pxTask == \old(pxCurrentTCB)) && !(
570                 \forall TASK t; FreeRTOS->state[t] == ready ==> FreeRTOS->priority[(TASK)
                        pxTask] >= FreeRTOS->priority[t]
571             )) ==> (FreeRTOS->state[(TASK) \old(pxCurrentTCB)] == ready &&
572                     FreeRTOS->state[(TASK) pxCurrentTCB] == running &&
573                     \old(FreeRTOS->state[(TASK) pxCurrentTCB]) == ready)
574         )
575         _(ensures \forall TASK t; (FreeRTOS->tasks[t] && \old(FreeRTOS->state[t]) !=
                FreeRTOS->state[t]) ==> transition[\old(FreeRTOS->state[t])][FreeRTOS->state[t
                ]])
576     {
577         _(assert \inv(FreeRTOS))
578         _(assert xSchedulerRunning == pdTRUE)
579         _(assert FreeRTOS->tasks[pxCurrentTCB])
580         _(assert pxTask != NULL ==> FreeRTOS->tasks[(TASK) pxTask])
581         _(assert \forall TASK t; FreeRTOS->tasks[t] ==> \inv(t))
582         _(assert \forall TASK t; FreeRTOS->tasks[t] ==> FreeRTOS->priority[t] <
                configMAX_PRIORITIES)
583         _(assert \forall TASK t; FreeRTOS->tasks[t] ==> FreeRTOS->state[t] <= 4)
584         _(assert pxCurrentTCB->\closed)
585         _(assert pxCurrentTCB \in FreeRTOS->\owns)
586         _(assert pxTask != NULL ==> ((TASK) pxTask)->\closed)
587         _(assert pxTask != NULL ==> ((TASK) pxTask) \in FreeRTOS->\owns)
588
589         tskTCB *pxTCB;
590         unsigned portBASE_TYPE uxCurrentPriority, uxPriorityUsedOnEntry;
591         portBASE_TYPE xYieldRequired = pdFALSE;
592
593         /* Ensure the new priority is valid. */
594         if( uxNewPriority >= configMAX_PRIORITIES )
595         {
596             uxNewPriority = configMAX_PRIORITIES - ( unsigned portBASE_TYPE ) 1U;
597         }
598         _(assert uxNewPriority < configMAX_PRIORITIES)
599
600         taskENTER_CRITICAL();
601         {
602             _(unwrapping FreeRTOS){
603                 _(assert \inv(FreeRTOS))
604                 _(assert \forall TASK t; (t->\closed && xSchedulerRunning != pdFALSE) ==>
605                     ((((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                            xTasksWaitingTermination ||
606                         t == NULL) <==> FreeRTOS->state[t] == nonexistent))
607                 _(atomic \embedding(&pxCurrentTCB)){
608                     if ( pxTask == pxCurrentTCB)
609                     {
610                         pxTask = NULL;
611                     }
612                 }
613
614                 _(atomic \embedding(&pxCurrentTCB)){
615                     pxTCB = pxTask != NULL ? (tskTCB *) pxTask
616                                            : pxCurrentTCB;
617                 }
618                 _(assume \forall TASK t; (t->\closed && xSchedulerRunning != pdFALSE) ==>
619                     ((((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                            xTasksWaitingTermination ||
620                         t == NULL) <==> FreeRTOS->state[t] == nonexistent))
621
622                 _(assert (TASK) pxTask == (TASK) pxCurrentTCB ==>
623                     pxTCB == pxCurrentTCB)
624
```

```
625                 _(assert \inv(FreeRTOS))
626                 _(assert \wrapped(pxCurrentTCB))
627                 _(assert pxTask != NULL ==> FreeRTOS->tasks[(TASK) pxTask])
628                 _(assert pxTask != NULL ==> \wrapped((TASK) pxTask))
629                 _(assert \wrapped(pxTCB))
630
631                 //#if ( configUSE_MUTEXES == 1 )
632                 //{
633                    // uxCurrentPriority = pxTCB->uxBasePriority;
634                 //}
635                 //#else
636                 //{
637                     uxCurrentPriority = pxTCB->uxPriority;
638                 //}
639                 //#endif
640
641                 if ( uxCurrentPriority != uxNewPriority )
642                 {
643                     /* The priority change may have readied a task of higher
644                     priority than the calling task. */
645
646                     if ( uxNewPriority > uxCurrentPriority )
647                     {
648                         if ( pxTask != NULL )
649                         {
650                             /* The priority of another task is being raised.  If we
651                             were raising the priority of the currently running task
652                             there would be no need to switch as it must have already
653                             been the highest priority task. */
654                             xYieldRequired = pdTRUE;
655                         }
656                     }
657                     else if ( pxTask == NULL )
658                     {
659                         /* Setting our own priority down means there may now be another
660                         task of higher priority that is ready to execute. */
661                         xYieldRequired = pdTRUE;
662                     }
663                 }
664
665                 uxPriorityUsedOnEntry = pxTCB->uxPriority;
666
667                 //#if ( configUSE_MUTEXES == 1 )
668                 //{{
669
670                 //  /* Only change the priority being used if the task is not
671                 //  currently using an inherited priority. */
672                 //  if( pxTCB->uxBasePriority == pxTCB->uxPriority )
673                 //  {
674                 //      pxTCB->uxPriority = uxNewPriority;
675                 //  }
676
677                 //  /* The base priority gets set whatever. */
678                 //  pxTCB->uxBasePriority = uxNewPriority;
679                 //}
680                 //#else
681                 //{
682                 _(assert \inv(FreeRTOS))
683                 _(unwrapping pxTCB){
684
685                     pxTCB->uxPriority = uxNewPriority;
686
687                     _(ghost FreeRTOS->priority[(TASK) pxTCB] = uxNewPriority)
688                     _(assert FreeRTOS->priority[(TASK) pxTCB] == pxTCB->uxPriority)
689                 //}
690                 //#endif
691                     /* If the task is in the blocked or suspended list we need do
692                     nothing more than change it's priority variable. However, if
693                     the task is in a ready list it needs to be removed and placed
694                     in the queue appropriate to its new priority. */
695                     //if( listIS_CONTAINED_WITHIN( &( pxReadyTasksLists[ uxCurrentPriority
                            ] ), &( pxTCB->xGenericListItem ) ) )
696                     if ( ( xList *) pxTCB->xGenericListItem.pvContainer == &(
                            pxReadyTasksLists[ uxCurrentPriority ] ))
697                     {
```

```
698                     //_(assert (xList *) pxTCB->xGenericListItem.pvContainer == &(
                            pxReadyTasksLists[ uxCurrentPriority ] ))
699                     //_(assert (xList *) pxTCB->xGenericListItem.pvContainer == &(
                            pxReadyTasksLists[ uxCurrentPriority ] ) <==>
700                     //    (FreeRTOS->state[(TASK) pxTCB] == running || FreeRTOS->state
                            [(TASK) pxTCB] == ready)
701                     //)
702                     _(assert FreeRTOS->state[(TASK) pxTCB] == running || FreeRTOS->
                            state[(TASK) pxTCB] == ready)
703                     /* The task is currently in its ready list - remove before adding
704                     it to it's new ready list.  As we are in a critical section we
705                     can do this even if the scheduler is suspended. */
706                     _(assert \wrapped(&(pxTCB->xGenericListItem)))
707                     if ( uxListRemove( ( xListItem * ) &( pxTCB->xGenericListItem ) )
                            == 0 )
708                     {
709                         //taskRESET_READY_PRIORITY( uxPriorityUsedOnEntry );
710                     }
711                     //prvAddTaskToReadyQueue( pxTCB );
712                     _(assert (&(pxTCB->xGenericListItem))->\closed)
713                     _(unwrapping &(pxTCB->xGenericListItem)){
714                         pxTCB->xGenericListItem.pvContainer = (xList *) &(
                                pxReadyTasksLists[pxTCB->uxPriority] );
715                     }
716
717                     _(assert FreeRTOS->tasks[(TASK) pxTCB])
718
719                     _(assert (xList *) pxTCB->xGenericListItem.pvContainer == &(
                            pxReadyTasksLists[ pxTCB->uxPriority ] ))
720                     _(assert FreeRTOS->state[(TASK) pxTCB] == running || FreeRTOS->
                            state[(TASK) pxTCB] == ready)
721                 }
722
723                 _(assert (FreeRTOS->state[(TASK) pxTCB] == running || FreeRTOS->state
                        [(TASK) pxTCB] == ready) ==>
724                     (xList *) pxTCB->xGenericListItem.pvContainer == &(
                            pxReadyTasksLists[ pxTCB->uxPriority ] ))
725
726                 if( xYieldRequired == pdTRUE )
727                 {
728                     _(assert FreeRTOS->tasks[topReady])
729
730                     _(assume FreeRTOS->state[topReady] == ready ||
731                         topReady == (TASK) pxCurrentTCB)
732                     _(assume \forall TASK t; (FreeRTOS->state[t] == ready || FreeRTOS
                            ->state[t] == running)  ==>
733                         FreeRTOS->priority[topReady] >= FreeRTOS->priority[t])
734                     _(assume FreeRTOS->priority[topReady] == FreeRTOS->priority[(TASK)
                             pxCurrentTCB]
735                         ==> topReady == (TASK) pxCurrentTCB)
736
737                     _(ghost {
738                         FreeRTOS->state[(TASK) pxCurrentTCB] = ready;
739                         FreeRTOS->state[topReady] = running;
740                     })
741
742                     //portYIELD_WITHIN_API();
743                     _(atomic \embedding(&pxCurrentTCB)){
744                         _(ghost pxCurrentTCB = topReady)
745                         _(bump_volatile_version \embedding(&pxCurrentTCB))
746                     }
747                     _(assert topReady != \old((TASK) pxCurrentTCB) ==> FreeRTOS->state
                            [(TASK) \old(pxCurrentTCB)] == ready)
748                 }
749
750             }//wrapping pxTCB
751             //_(assert \false)
752
753             _(assume \forall TASK t; (FreeRTOS->tasks[t] && ((tskTCB *)t != pxTCB) &&
                    xSchedulerRunning != pdFALSE) ==>
754                 (
755                     (((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                            pxReadyTasksLists[((tskTCB *) t)->uxPriority] &&
756                     ((tskTCB *) t) != pxCurrentTCB)
757                         <==> FreeRTOS->state[t] == ready
```

```
758                     )
759                 )
760
761                 _(assume \forall TASK t; (FreeRTOS->tasks[t] && ((tskTCB *)t != pxTCB) &&
                        xSchedulerRunning != pdFALSE) ==>
762                     (
763                         (((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                                xDelayedTaskList1 ||
764                             ((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                                    xDelayedTaskList2)
765                             <==> FreeRTOS->state[t] == blocked
766                     )
767                 )
768
769                 _(assume \forall TASK t; (FreeRTOS->tasks[t] && ((tskTCB *)t != pxTCB) &&
                        xSchedulerRunning != pdFALSE) ==>
770                     (
771                         ((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                                xSuspendedTaskList
772                             <==> FreeRTOS->state[t] == suspended
773                     )
774                 )
775
776                 _(assume \forall TASK t; (t->\closed && ((tskTCB *)t != pxTCB) &&
                        xSchedulerRunning != pdFALSE) ==>
777                     (
778                         (((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                                xTasksWaitingTermination ||
779                             t == NULL)
780                             <==> FreeRTOS->state[t] == nonexistent
781                     )
782                 )
783
784             }//wrapping FreeRTOS
785
786             _(assert \inv(FreeRTOS))
787         }
788         taskEXIT_CRITICAL();
789         _(assume Trans())
790     }
791
792 #endif
793
794
795 #if ( INCLUDE_vTaskSuspend == 1 )
796
797     void vTaskSuspend( xTaskHandle pxTaskToSuspend _(ghost FRTOS FreeRTOS) _(ghost TASK
            topReady))
798     _(updates FreeRTOS)
799     _(requires \mutable(&xSchedulerRunning))
800     _(requires xSchedulerRunning == pdTRUE)
801     _(requires excList())
802
803     _(requires FreeRTOS->tasks[(TASK) pxTaskToSuspend])
804     _(requires (tskTCB *)pxTaskToSuspend != (tskTCB *)xIdleTaskHandle)
805
806     _(requires FreeRTOS->tasks[topReady])
807     _(requires FreeRTOS->state[topReady] == ready)
808     _(requires \forall TASK rts;
809         (FreeRTOS->tasks[rts] && FreeRTOS->state[rts] == ready)
810             ==> FreeRTOS->priority[topReady] >= FreeRTOS->priority[rts])
811     _(requires topReady != (TASK) pxTaskToSuspend)
812     _(requires topReady != (TASK) pxCurrentTCB)
813
814     _(ensures FreeRTOS->state[(TASK) pxTaskToSuspend] == suspended)
815     _(ensures pxTaskToSuspend == NULL ==> (TASK) pxTaskToSuspend == \old(pxCurrentTCB))
816     _(ensures pxTaskToSuspend == NULL ==> (TASK) pxCurrentTCB == topReady)
817     _(ensures \forall TASK t; (FreeRTOS->tasks[t] && \old(FreeRTOS->state[t]) != FreeRTOS
            ->state[t]) ==> transition[\old(FreeRTOS->state[t])][FreeRTOS->state[t]])
818     {
819     _(assert \wrapped(FreeRTOS))
820     _(assert \inv(FreeRTOS))
821     _(assert xSchedulerRunning == pdTRUE)
822     //_(assert FreeRTOS->tasks[(TASK) pxTaskToDelete])
823     //_(assert pxCurrentTCB \in FreeRTOS->\owns)
```

```
824     _(assert \forall TASK t; FreeRTOS->tasks[t] ==> \inv(t))
825     _(assert \forall TASK t; FreeRTOS->tasks[t] ==> FreeRTOS->priority[t] <
            configMAX_PRIORITIES)
826     _(assert \forall TASK t; FreeRTOS->tasks[t] ==> FreeRTOS->state[t] <= 4)
827     _(assert pxCurrentTCB->\closed)
828     _(assert ((TASK)pxTaskToSuspend)->\closed)
829
830     tskTCB *pxTCB;
831
832     _(unwrapping FreeRTOS){
833         taskENTER_CRITICAL();
834         {
835             _(assert \inv(FreeRTOS))
836             /* Ensure a yield is performed if the current task is being
837             suspended. */
838             if( pxTaskToSuspend == _(atomic_read \embedding(&pxCurrentTCB))
839                                    pxCurrentTCB )
840             {
841                 pxTaskToSuspend = NULL;
842             }
843
844             /* If null is passed in here then we are suspending ourselves. */
845             pxTCB = _(atomic_read \embedding(&pxCurrentTCB))
846                 prvGetTCBFromHandle( pxTaskToSuspend );
847             _(assert (TASK) pxTaskToSuspend == (TASK) pxCurrentTCB ==>
848                     pxTCB == pxCurrentTCB)
849             _(assert \wrapped(pxTCB))
850
851             _(unwrapping pxTCB){
852                 /* Remove task from the ready/delayed list and place in the suspended list
                        . */
853                 _(assert \wrapped(&(pxTCB->xGenericListItem)))
854                 if( uxListRemove( ( xListItem * ) &( pxTCB->xGenericListItem ) ) == 0 )
855                 {
856                     //taskRESET_READY_PRIORITY( pxTCB->uxPriority );
857                 }
858
859                 _(assert pxTCB->xGenericListItem.pvContainer == NULL)
860
861                 /* Is the task waiting on an event also? */
862                 if( pxTCB->xEventListItem.pvContainer != NULL )
863                 {
864                     uxListRemove( &( pxTCB->xEventListItem ) );
865                 }
866
867                 _(assert pxTCB->xEventListItem.pvContainer == NULL)
868
869                 vListInsertEnd( ( xList * ) &xSuspendedTaskList, &( pxTCB->
                        xGenericListItem ) );
870                 _(assert (xList *)(pxTCB->xGenericListItem.pvContainer) == (xList *) &
                        xSuspendedTaskList)
871
872             }
873
874             _(ghost {
875                 FreeRTOS->state[(TASK) pxTCB] = suspended;
876             })
877         }
878         taskEXIT_CRITICAL();
879
880         _(assert \forall TASK t; (FreeRTOS->tasks[t] && ((tskTCB *)t != pxTCB) &&
                xSchedulerRunning != pdFALSE) ==>
881             (
882                 (((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                        pxReadyTasksLists[((tskTCB *) t)->uxPriority] &&
883                 ((tskTCB *) t) != pxCurrentTCB)
884                     <==> FreeRTOS->state[t] == ready
885             )
886         )
887
888         _(assert \forall TASK t; (FreeRTOS->tasks[t] && ((tskTCB *)t != pxTCB) &&
                xSchedulerRunning != pdFALSE) ==>
889             (
890                 (((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                        xDelayedTaskList1 ||
```

```
891                     ((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                            xDelayedTaskList2)
892                     <==> FreeRTOS->state[t] == blocked
893                 )
894             )
895
896             _(assert \forall TASK t; (FreeRTOS->tasks[t] && ((tskTCB *)t != pxTCB) &&
                    xSchedulerRunning != pdFALSE) ==>
897                 (
898                     ((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                            xSuspendedTaskList
899                     <==> FreeRTOS->state[t] == suspended
900                 )
901             )
902
903             if( ( void * ) pxTaskToSuspend == NULL )
904             {
905                 if( xSchedulerRunning != pdFALSE )
906                 {
907                     /* We have just suspended the current task. */
908                     //portYIELD_WITHIN_API();
909                     _(atomic \embedding(&pxCurrentTCB)){
910                         _(ghost pxCurrentTCB = topReady)
911                         _(bump_volatile_version \embedding(&pxCurrentTCB))
912                     }
913                     _(ghost FreeRTOS->state[topReady] = running)
914
915                     _(assert FreeRTOS->state[(TASK) pxTCB] ==suspended)
916                     _(assert (xList *)(pxTCB->xGenericListItem.pvContainer) == &
                            xSuspendedTaskList)
917
918                     _(assert FreeRTOS->state[(TASK) pxCurrentTCB] == running)
919                     _(assert \old(pxCurrentTCB) == pxTCB)
920                     _(assert FreeRTOS->state[(TASK) \old(pxCurrentTCB)] == suspended)
921                 }
922                 //else
923                 //{
924                 //  /* The scheduler is not running, but the task that was pointed
925                 //  to by pxCurrentTCB has just been suspended and pxCurrentTCB
926                 //  must be adjusted to point to a different task. */
927                 //  if( listCURRENT_LIST_LENGTH( &xSuspendedTaskList ) ==
                        uxCurrentNumberOfTasks )
928                 //  {
929                 //      /* No other tasks are ready, so set pxCurrentTCB back to
930                 //      NULL so when the next task is created pxCurrentTCB will
931                 //      be set to point to it no matter what its relative priority
932                 //      is. */
933                 //      pxCurrentTCB = NULL;
934                 //  }
935                 //  else
936                 //  {
937                 //      vTaskSwitchContext();
938                 //  }
939                 //}
940             }
941
942             _(assert pxTaskToSuspend != NULL ==> FreeRTOS->state[topReady] == ready)
943
944             _(assume \forall TASK t; (FreeRTOS->tasks[t] && ((tskTCB *)t != pxTCB) &&
                    xSchedulerRunning != pdFALSE) ==>
945                 (
946                     (((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                            pxReadyTasksLists[((tskTCB *) t)->uxPriority] &&
947                     ((tskTCB *) t) != pxCurrentTCB)
948                         <==> FreeRTOS->state[t] == ready
949                 )
950             )
951
952             _(assume \forall TASK t; (FreeRTOS->tasks[t] && ((tskTCB *)t != pxTCB) &&
                    xSchedulerRunning != pdFALSE) ==>
953                 (
954                     (((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                            xDelayedTaskList1 ||
955                         ((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                                xDelayedTaskList2)
```

```
956                     <==> FreeRTOS->state[t] == blocked
957                 )
958             )
959
960             _(assume \forall TASK t; (FreeRTOS->tasks[t] && ((tskTCB *)t != pxTCB) &&
                    xSchedulerRunning != pdFALSE) ==>
961                 (
962                     ((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                        xSuspendedTaskList
963                     <==> FreeRTOS->state[t] == suspended
964                 )
965             )
966
967             _(assume \forall TASK t; (t->\closed && ((tskTCB *)t != pxTCB) &&
                    xSchedulerRunning != pdFALSE) ==>
968                 (
969                     (((xList *)((tskTCB *) t)->xGenericListItem.pvContainer) == &
                        xTasksWaitingTermination ||
970                     t == NULL)
971                     <==> FreeRTOS->state[t] == nonexistent
972                 )
973             )
974
975             _(assert \forall TASK t; FreeRTOS->tasks[t] ==> t->\closed)
976             }
977
978             _(assume Trans())
979         }
980
981     #endif
982
983     #if ( INCLUDE_vTaskSuspend == 1 )
984
985         void vTaskResume( xTaskHandle pxTaskToResume _(ghost FRTOS FreeRTOS) )
986             _(updates FreeRTOS)
987             _(requires xSchedulerRunning == pdTRUE)
988             _(requires excList())
989
990             _(requires FreeRTOS->tasks[(TASK) pxTaskToResume])
991             _(requires FreeRTOS->state[(TASK) pxTaskToResume] == suspended)
992
993             _(ensures ((TASK) pxTaskToResume)->\closed)
994             _(ensures pxCurrentTCB->\closed)
995             _(ensures FreeRTOS->tasks[(TASK) pxTaskToResume])
996             _(ensures FreeRTOS->tasks[(TASK) pxCurrentTCB])
997             _(ensures ((tskTCB *)pxTaskToResume)->uxPriority > \old(pxCurrentTCB)->uxPriority
                    ==> (FreeRTOS->state[(TASK)\old(pxCurrentTCB)]) == ready && (TASK)
                    pxCurrentTCB == (TASK) pxTaskToResume && (FreeRTOS->state[(TASK)pxCurrentTCB])
                     == running)
998             _(ensures ((tskTCB *)pxTaskToResume)->uxPriority <= \old(pxCurrentTCB)->uxPriority
                     ==> (FreeRTOS->state[(TASK) pxTaskToResume]) == ready && FreeRTOS->state[(
                    TASK)\old(pxCurrentTCB)] == running && \old(pxCurrentTCB) == pxCurrentTCB )
999             _(ensures \forall TASK t; (FreeRTOS->tasks[t] && \old(FreeRTOS->state[t]) !=
                    FreeRTOS->state[t]) ==> transition[\old(FreeRTOS->state[t])][FreeRTOS->state[t
                    ]])
1000        {
1001            _(assert \wrapped(FreeRTOS))
1002            _(assert \inv(FreeRTOS))
1003            _(assert xSchedulerRunning == pdTRUE)
1004            _(assert \forall TASK t; FreeRTOS->tasks[t] ==> FreeRTOS->state[t] <= 4)
1005            _(assert ((TASK) pxTaskToResume)->\closed)
1006            _(assert ((TASK) pxTaskToResume) \in FreeRTOS->\owns)
1007            _(assert ((TASK) pxCurrentTCB)->\closed)
1008            _(assert ((TASK) pxCurrentTCB) \in FreeRTOS->\owns)
1009            _(assert FreeRTOS->tasks[(TASK) pxTaskToResume])
1010            _(assert FreeRTOS->tasks[(TASK) pxCurrentTCB])
1011            _(assert (TASK) pxTaskToResume != (TASK) pxCurrentTCB)
1012
1013            _(assert \forall TASK t; FreeRTOS->tasks[t] ==> \inv(t))
1014
1015        tskTCB *pxTCB;
1016
1017            /* It does not make sense to resume the calling task. */
1018            //configASSERT( pxTaskToResume );
1019
```

```
1020            /* Remove the task from whichever list it is currently in, and place
1021            it in the ready list. */
1022            pxTCB = ( tskTCB * ) pxTaskToResume;
1023
1024            _(assert pxTCB != pxCurrentTCB)
1025            _(assert pxTCB == (tskTCB *) pxTaskToResume)
1026
1027            /* The parameter cannot be NULL as it is impossible to resume the
1028            currently executing task. */
1029            _(assert pxCurrentTCB->\closed)
1030
1031
1032            _(unwrapping FreeRTOS){
1033                _(assert \wrapped(pxCurrentTCB))
1034                _(assert \inv(FreeRTOS))
1035
1036                if( ( pxTCB != NULL ) && ( pxTCB != _(atomic_read \embedding(&pxCurrentTCB))
                        pxCurrentTCB ) )
1037                {
1038                    _(assume \inv(FreeRTOS))
1039
1040                    taskENTER_CRITICAL();
1041                    {
1042                        _(assert \inv(FreeRTOS))
1043                        _(assert FreeRTOS->tasks[(TASK) pxTCB])
1044                        _(assert FreeRTOS->tasks[pxCurrentTCB])
1045                        _(assert \wrapped((TASK) pxTCB))
1046                        _(assert \wrapped((TASK) pxCurrentTCB))
1047                        _(unwrapping pxTCB){
1048                            //if( xTaskIsTaskSuspended( pxTCB ) == pdTRUE )
1049                            if (((xList *)(pxTCB)->xGenericListItem.pvContainer) == &
                                    xSuspendedTaskList)
1050                            {
1051                                _(assert FreeRTOS->state[(TASK) pxTCB] == suspended)
1052                                //traceTASK_RESUME( pxTCB );
1053                                    /* As we are in a critical section we can access the ready
1054                                    lists even if the scheduler is suspended. */
1055                                    _(assert \wrapped(&(pxTCB->xGenericListItem)))
1056                                    uxListRemove(  &( pxTCB->xGenericListItem ) );
1057                                    _(ghost FreeRTOS->state[(TASK) pxTCB] = ready)
1058                                    //prvAddTaskToReadyQueue( pxTCB );
1059
1060                                    _(assert \wrapped(pxCurrentTCB))
1061                                    /* We may have just resumed a higher priority task. */
1062                                    if( pxTCB->uxPriority >= _(atomic_read \embedding(&
                                            pxCurrentTCB)) pxCurrentTCB->uxPriority )
1063                                    {
1064                                        _(assert FreeRTOS->priority[(TASK) pxTCB] == pxTCB->
                                                uxPriority)
1065                                        /* This yield may not cause the task just resumed to
                                                run, but
1066                                        will leave the lists in the correct state for the next
                                                yield. */
1067                                        //portYIELD_WITHIN_API();
1068                                        _(atomic \embedding(&pxCurrentTCB)){
1069                                            _(ghost pxCurrentTCB = pxTCB)
1070                                            _(bump_volatile_version \embedding(&pxCurrentTCB))
1071                                        }
1072
1073                                    }
1074
1075                            }
1076                        }
1077                    }
1078                    _(assume \inv(FreeRTOS))
1079                }
1080                taskEXIT_CRITICAL();
1081                _(assume Trans())
1082            }
1083        }
1084
1085    #endif
```

Bibliography

[1] E. M. Clarke and J. M. Wing, “Formal methods: state of the art and future directions,” *ACM Comput. Surv.*, vol. 28, pp. 626–643, December 1996.

[2] J.-M. Jazequel and B. Meyer, “Design by contract: the lessons of Ariane,” *Computer*, vol. 30, no. 1, pp. 129 –130, Jan 1997.

[3] C. A. R. Hoare and J. Misra, “Verified software: Theories, tools, experiments vision of a grand challenge project,” in *Verified Software: Theories, Tools, Experiments*, ser. Lecture Notes in Computer Science, B. Meyer and J. Woodcock, Eds. Springer Berlin / Heidelberg, 2008, vol. 4171, pp. 1–18.

[4] J. Woodcock, “First steps in the Verified Software Grand Challenge,” *IEEE Computer*, vol. 39, no. 10, pp. 57–64, 2006.

[5] C. Jones, P. O’Hearn, and J. Woodcock, “Verified software: a grand challenge,” *IEEE Computer*, vol. 39, no. 4, pp. 93–95, 2006.

[6] A. Hall, “Seven myths of formal methods,” *Software, IEEE*, vol. 7, no. 5, pp. 11 –19, Sep. 1990.

[7] S. King *et al.*, “Is proof more cost-effective than testing?” *Software Engineering, IEEE Transactions on*, vol. 26, no. 8, pp. 675 –686, Augest 2000.

[8] V. Casey, *Software Testing and Global Industry: Future Paradigms*. United Kingdom: Cambridge Scholars Publishing, 2009.

[9] E. W. Dijkstra, “Notes on structured programming,” in *Structured programming*, O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Eds. London, UK, UK: Academic Press Ltd., 1972, pp. 3–6.

[10] J. Bowen, *Formal Specification and Documentation Using Z: A Case Study Approach*. London, UK: International Thomson Computer Press, 2003.

[11] J. Woodcock *et al.*, “Formal methods: Practice and experience,” *ACM Comput. Surv.*, vol. 41, pp. 19:1–19:36, October 2009.

[12] G. Berry, “Synchronous design and verification of critical embedded systems

using SCADE and Esterel," in *Formal Methods for Industrial Critical Systems*, ser. Lecture Notes in Computer Science, S. Leue and P. Merino, Eds. Springer Berlin / Heidelberg, 2008, vol. 4916, pp. 2–2.

[13] C. A. R. Hoare, "The verifying compiler: a grand challenge for computing research," *Journal of the ACM*, vol. 50, no. 1, pp. 63–69, 2003.

[14] J. Bicarregui, C. Hoare, and J. Woodcock, "The verified software repository: a step towards the verifying compiler," *Formal Aspects of Computing*, vol. 18, pp. 143–151, 2006.

[15] C. A. R. Hoare *et al.*, "The Verified Software Initiative: a manifesto," *ACM Comput. Surv.*, vol. 41, no. 4, 2009.

[16] J. Woodcock *et al.*, "The certification of the Mondex electronic purse to ITSEC Level E6," *Formal Aspects of Computing*, vol. 20, no. 1, pp. 5–19, 2008.

[17] L. Freitas, Z. Fu, and J. Woodcock, "POSIX file store in Z/Eves: an experiment in the verified software repository," in *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*, 2007, pp. 3–14.

[18] R. Barry, "Using the FreeRTOS Real Time Kernel—A Practical Guide," PDF book available from www.freertos.org, 2012.

[19] R. Barry, "FreeRTOS Reference Manual—API functions and configuration options," PDF book available from shop.freertos.org, 2012.

[20] E. Cohen *et al.*, "VCC: A Practical System for Verifying Concurrent C," in *Theorem Proving in Higher Order Logics*, ser. Lecture Notes in Computer Science, S. Berghofer *et al.*, Eds. Springer Berlin Heidelberg, 2009, vol. 5674, pp. 23–42. [Online]. Available: DOI:10.1007/978-3-642-03359-9_2

[21] M. Moskal *et al.*, *Verifying C Programs: A VCC Tutorial*, MSR Redmond, EMIC, 2012.

[22] M. Barnett *et al.*, "Boogie: A modular reusable verifier for object-oriented programs," in *Formal Methods for Components and Objects*, ser. Lecture Notes in Computer Science, F. Boer *et al.*, Eds. Springer Berlin, Heidelberg, 2006, vol. 4111, pp. 364–387. [Online]. Available: DOI:10.1007/11804192_17

[23] L. Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, C. Ramakrishnan and J. Rehof, Eds. Springer Berlin Heidelberg, 2008, vol. 4963, pp. 337–340. [Online]. Available: DOI:10.1007/978-3-540-78800-3_24

[24] Intel Corporation. (undated) Intel Museum – The Intel 4004. [Online]. Available: www.intel.com/about/companyinfo/museum/exhibits/4004/facts.htm

[25] G. Moore, “Cramming more components onto integrated circuits,” *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82 –85, Jan. 1998.

[26] Intel Corporation. (undated) Intel® Core TM i7-680UM Processor (4m cache, 1.46 ghz) with spec code(s)slbst. [Online]. Available: ark.intel.com/Product.aspx?id=49664&processor=i7-680UM&spec-codes=SLBST

[27] P. Bose, D. Albonesi, and D. Marculescu, “Guest editors’ introduction: Power and complexity aware design,” *Micro, IEEE*, vol. 23, no. 5, pp. 8–11, Sept.–Oct. 2003.

[28] M. B. Taylor *et al.*, “Tiled multicore processors,” in *Multicore Processors and Systems*, S. W. Keckler, K. Olukotun, and H. P. Hofstee, Eds. Springer US, 2009, pp. 1–33.

[29] R. Camposano and J. Wilberg, “Embedded system design,” *Design Automation for Embedded Systems*, vol. 1, pp. 5–50, 1996.

[30] Intel Research. (undated) Teraflops research chip. [Online]. Available: techresearch.intel.com/ProjectDetails.aspx?Id=151

[31] Intel Corporation. (undated) Intel® Pentium® 4 Processor 531 supporting HT Technology (1M Cache, 3.00 GHz, 800 MHz FSB). [Online]. Available: ark.intel.com/Product.aspx?id=27465

[32] I. Meisels and M. Saaltink, *Z/Eves 1.5 Reference Manual*, ORA Canada, September 1997, tR-97-5493-03d.

[33] ProB Documentation. (2010) The ProB Animator and Model Checker. [Online]. Available: www.stups.uni-duesseldorf.de/ProB/index.php5/Main_Page

[34] Real Time Engineers Ltd. (undated) The FreeRTOS project. [Online]. Available: www.freertos.org

[35] L. G. Valiant, “A bridging model for multi-core computing,” in *Proceedings of the 16th Annual European Symposium on Algorithms*, ser. ESA ’08, 2008, pp. 13–28.

[36] Real Time Engineers Ltd. (undated) Coding Standard and Style Guide. [Online]. Available: http://www.freertos.org/FreeRTOS-Coding-Standard-and-Style-Guide.html

[37] S. Cheng, J. Woodcock, and D. D’Souza, “Using formal reasoning on a model of tasks for FreeRTOS,” *Formal Aspects of Computing*, vol. 27, no. 1, pp. 167–192, 2015. [Online]. Available: DOI:10.1007/s00165-014-0308-9

[38] I. D. Craig, *Formal Models of Operating System Kernels*. Springer, 2006.

[39] I. D. Craig, *Formal Refinement for Operating System Kernels*. Springer, 2007.

[40] J. J. Labrosse, *MicroC OS II: The Real Time Kernel*. Newnes, 2002.

[41] L. Freitas and J. Woodcock, "A chain datatype in Z," *Int. J. Software and Informatics*, vol. 3, no. 2–3, pp. 357–374, 2009.

[42] E. Börger and I. Craig, "Modeling an operating system kernel," in *Informatik als Dialog zwischen Theorie und Anwendung*, V. Diekert, K. Weicker, and N. Weicker, Eds. Vieweg+Teubner, 2009, pp. 199–216. [Online]. Available: DOI:10.1007/978-3-8348-9982-8_17

[43] G. Klein, "Operating system verification—an overview," *Sādhanā*, vol. 34, no. 1, pp. 27–69, Feb. 2009.

[44] G. Klein, "The L4.verified project—next steps," in *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE 2010, Edinburgh, UK, August 16-19, 2010. Proceedings*, ser. Lecture Notes in Computer Science, G. T. Leavens, P. W. O'Hearn, and S. K. Rajamani, Eds., vol. 6217. Springer, 2010, pp. 86–96.

[45] G. Klein, "A formally verified OS kernel. Now what?" in *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11–14, 2010. Proceedings*, ser. Lecture Notes in Computer Science, M. Kaufmann and L. C. Paulson, Eds., vol. 6172. Springer, 2010, pp. 1–7.

[46] G. Klein, "From a verified kernel towards verified systems," in *Programming Languages and Systems—8th Asian Symposium, APLAS 2010, Shanghai, China, November 28–December 1, 2010. Proceedings*, ser. Lecture Notes in Computer Science, K. Ueda, Ed., vol. 6461. Springer, 2010, pp. 21–33.

[47] T. Sewell *et al.*, "seL4 enforces integrity," in *Interactive Theorem Proving—Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22–25, 2011. Proceedings*, ser. Lecture Notes in Computer Science, M. C. J. D. van Eekelen *et al.*, Eds., vol. 6898. Springer, 2011, pp. 325–340. [Online]. Available: DOI:10.1007/978-3-642-22863-6_24

[48] R. Buerki and A.-K. Rueegsegger, "Muen–an x86/64 separation kernel for high assurance," University of Applied Sciences Rapperswil (HSR), Tech. Rep., 2013.

[49] D. Déharbe, S. Galvão, and A. M. Moreira, "Formalizing FreeRTOS: first steps," in *Formal Methods: Foundations and Applications, 12th Brazilian Symposium on Formal Methods, SBMF 2009, Gramado, Brazil, August 19–21, 2009, Revised Selected Papers*, ser. Lecture Notes in Computer Science, M. V. M. Oliveira and J. Woodcock, Eds., vol. 5902. Springer, 2009, pp. 101–117. [Online]. Available: DOI:10.1007/978-3-642-10452-7_8

[50] C. Pronk, “Verifying FreeRTOS: a feasibility study,” Delft University of Technology, Software Engineering Research Group, Tech. Rep. TUD-SERG-2010-042, 2010.

[51] Y. Lin, “Formal Analysis of FreeRTOS,” Master’s thesis, University of York, 2010.

[52] R. Arthan, “ProofPower,” Web page www.lemma-one.com/ProofPower/, 2012.

[53] L. Paulson, “Isabelle,” Web page www.cl.cam.ac.uk/research/hvg/isabelle/, 2012.

[54] S. Owre, N. Shankar, and J. Rushby, “PVS,” Web page: pvs.csl.sri.com, 2012.

[55] J. T. Muehlberg and L. Freitas, “Verifying FreeRTOS: from requirements to binary code,” in *Proceedings of the 11th International Workshop on Automated Verification of Critical Systems (AVoCS 2011)*, ser. Electronic Communications of the EASST, J. Bendisposto *et al.*, Eds., vol. X, 2011, pp. 1–2.

[56] J. T. Mühlberg and G. Lüttgen, “Symbolic object code analysis,” in *Model Checking Software—17th International SPIN Workshop, Enschede, The Netherlands, September 27–29, 2010. Proceedings*, ser. Lecture Notes in Computer Science, J. van de Pol and M. Weber, Eds., vol. 6349. Springer, 2010, pp. 4–21.

[57] B. Jacobs, J. Smans, and F. Piessens, “A quick tour of the VeriFast program verifier,” in *Proceedings of the 8th Asian Conference on Programming Languages and Systems*, ser. APLAS’10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 304–311. [Online]. Available: dl.acm.org/citation.cfm?id=1947873.1947902

[58] J. Ferreira, G. He, and S. Qin, “Automated verification of the FreeRTOS scheduler in HIP/SLEEK,” in *6th International Symposium on Theoretical Aspects of Software Engineering (TASE’12)*, 4–6 July 2012.

[59] W. Su and J.-R. Abrial, May 2011, private communication.

[60] J. Mistry, “FreeRTOS and multicore,” September 2011, MSc Dissertation, Department of Computer Science, University of York.

[61] J. Mistry, M. Naylor, and J. Woodcock, “Adapting FreeRTOS for multicores: an experience report,” *Software: Practice and Experience*, vol. 44, no. 9, pp. 1129–1154, 2014. [Online]. Available: DOI:10.1002/spe.2188

[62] S. Divakaran, D. D’Souza, and N. Sridhar, “Efficient refinement checking in VCC,” in *Verified Software: Theories, Tools and Experiments*, ser. Lecture Notes in Computer Science, D. Giannakopoulou and D. Kroening, Eds. Springer International Publishing, 2014, vol. 8471, pp. 21–36. [Online].

Available: http://dx.doi.org/10.1007/978-3-319-12154-3_2

[63] I. Sørensen, “A specification language,” in *Program Specification*, ser. Lecture Notes in Computer Science, J. Staunstrup, Ed. Springer Berlin / Heidelberg, 1982, vol. 134, pp. 381–401.

[64] I. Toyn, Ed., *Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics*, 1st ed., ser. ISO/ICE 13568:2002. ISO, Oct. 2002.

[65] J. Woodcock and J. Davies, *Using Z: Specification, Refinement, and Proof*, ser. International Series in Computer Science. Prentice-Hall, 1996.

[66] J. M. Spivey, *The Z Notation: A Reference Manual*, 2nd ed., ser. Series in Computer Science. Prentice Hall International, 1992.

[67] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Commun. ACM*, vol. 12, pp. 576–580, October 1969.

[68] J. M. Spivey. (1990) A guide to the zed style option. [Online]. Available: www.ctan.org/tex-archive/macros/latex/contrib/zed-csp/zed2e.pdf

[69] M. Saaltink, *Z/Eves 2.0 User’s Guide*, ORA Canada, 1999, TR-99-5493-06a.

[70] M. Saaltink, *Z/Eves 2.0 Mathematical Toolkit*, ORA Canada, October 1999, TR-99-5493-05b.

[71] G. Klein *et al.*, “seL4: formal verification of an OS kernel,” in *SOSP*, 2009, pp. 207–220.

[72] Community Z Tools Project. (2013, November) Community Z Tools. [Online]. Available: czt.sourceforge.net

[73] T. P. Baker, “A comparison of global and partitioned EDF schedulability tests for multiprocessors,” in *Proceedings of the International Conference on Real-Time and Network Systems*, 2006, pp. 119–130.

[74] M. R. Garey, R. L. Graham, and J. D. Ullman, “Worst-case analysis of memory allocation algorithms,” in *Proceedings of the fourth annual ACM symposium on Theory of computing*, ser. STOC ’72. New York, NY, USA: ACM, 1972, pp. 143–150. [Online]. Available: DOI:10.1145/800152.804907