



Tim Sagaster, BSc, BSc

Verification of Real-Time Resource Management Protocols – From Code to Model

MASTER'S THESIS

to achieve the university degree of
Diplom-Ingenieur

submitted to

Graz University of Technology

Supervisor

Dipl.-Ing. Tobias Scheipel, BSc
Institute of Technical Informatics
Embedded Automotive Systems Group

Graz, January 2024

*We are stuck with technology when what we really want is just
stuff that works. – Douglas Adams*

Acknowledgements

This thesis would not have been possible without the support of several people.

First and foremost, I want to thank my supervisor, Tobias Scheipel, who helped me tremendously during the time I wrote the thesis, often giving me valuable feedback on the same day I sent him new chapters, and whose boundless support made this thesis possible.

Without the guidance of Professor Marcel Baunach I would not have been able to get to the point where I could make this thesis become reality. I am incredibly thankful to him and his whole team in the Embedded Automotive Systems group for the time I was able to spend there. A special mention goes out to Leandro Batista Ribeiro, who helped me whenever I encountered issues while working on the UPPAAL models.

Special thanks go to all the members of the Aerospace Team Graz who, even though it substantially prolonged my studies, inspired me to venture into embedded systems and make contact with the world of Real-Time Operating Systems.

I am very grateful for the patience my colleagues at the European Space Agency had with me while I was finishing the thesis and, therefore, might not have been completely focused on my work at all times.

I want to thank all my colleagues and friends I met along the journey of studying at the Graz University of Technology, without whom I would not have made it to the end.

Lastly, I would like to express my gratitude to my family and friends who accompanied me along all my different journeys, bringing so much joy into my life.

Graz, January 2024

Tim Sagaster

Abstract

Current verification and validation methods of safety-critical code are still primarily based on testing and manual reviews. This is a time-consuming and costly process that depends on the availability of experts and still does not guarantee the correctness of the source code.

This thesis examines the automatic translation of C code to a model of the modelling tool UPPAAL, which can further be used for formal verification. To obtain a suitable code base, three different resource management protocols for the Real-Time Operating System *SmartOS* are defined and implemented. The code is then manually translated, resulting in models consisting of graphs representing the control flow and C-like code updating the data structures of the models.

The aim of the thesis is to investigate if this translation can be automated by condensing the translation process to basic rules. This process worked well for translating the C code to the C-like code used in the model but fell short of creating the graphs, which proved to be unsuitable for automation.

The resulting model was further used to improve the source code quality of *SmartOS* and to find bugs not found during the implementation testing, emphasizing the benefits an accompanying model can have on the correctness of source code. Verifying the implementation itself is not in the scope of this thesis, as this would require additional proof of the one-to-one correlation between code and model.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Research Question and Objectives	2
1.3	Thesis Structure and Organisation	3
2	Background and Terminology	5
2.1	SmartOS	5
2.1.1	Tasks	5
2.1.2	Events	6
2.1.3	Resources	6
2.1.4	Scheduling	6
2.2	UPPAAL	7
2.2.1	Automata	7
2.2.2	Parameters	9
2.2.3	Declarations	9
3	Related Work	11
4	Definition of Protocols	13
4.1	Common Features	13
4.2	Highest Locker Protocol (HLP)	14
4.3	Priority Ceiling Protocol (PCP)	15
4.4	Priority Inheritance Protocol (PIP)	16
4.5	Example	18
5	From Code to Model	21
5.1	Code Implementation	21
5.1.1	Directory Structure	21
5.1.2	Including the Chosen Resource Management Protocol	23

5.1.3	Resource Management Interface	23
5.2	Model	26
5.2.1	Base Model	26
5.2.2	HLP	31
5.2.3	PCP	35
5.2.4	PIP	39
5.3	Transforming Code	43
5.3.1	Pointers	43
5.3.2	Types	44
5.3.3	Data Structures	45
5.3.4	Functions	51
6	Conclusion and Future Work	59
6.1	Answering the Research Question	59
6.2	Secondary Goals	60
6.3	Future Work	60
	Bibliography	63
	List of Figures	70
	List of Tables	71
	List of Listings	75
	List of Abbreviations	77

CHAPTER 1

Introduction

1.1 Motivation

The focus in the safety-critical systems domain has long moved from only mechanical systems to also include electronics and software. With the advent of more complex embedded software systems that require more than a simple main loop, the use of Real-Time Operating Systems (RTOSs) has become the norm. Ensuring that these RTOSs fulfil the required safety standards is not easy, and different approaches have been used in different fields. The automotive domain in Europe has opted to go to a standardized system with the introduction of first the "Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug" (OSEK) standard [1] and later the "AUTomotive Open System ARchitecture" (AUTOSAR) Standard [2]. These standards do not directly come with an implementation but merely define the interfaces of the software, relying on commercial vendors for the implementation. AUTOSAR comes with acceptance tests [3] to qualify a vendor's implementation. Another example comes from the civil aerospace industry, where the European Space Agency (ESA) has released a pre-qualified version of Real-Time Executive for Multiprocessor Systems (RTEMS) [4], an open source RTOS, which consists of pre-compiled versions of the RTOS for the GR712RC [5] and GR740 [6], two of the most common processors used in ESA missions, eliminating the need to requalify this part of the software for every mission. For the use of other RTOSs, ESA requires qualification using the European Cooperation for Space Standardization (ECSS) standards, considering the RTOS part of the application software.

Both of those approaches use testing as the baseline of their qualification. As test cases can never guarantee to have 100% coverage of all possible situations, the need for a more robust method arises. Model-based approaches try to fill this need, as they promise to verify the

complete system formally. The problem with this approach is that one needs to translate the model into code without introducing new bugs. Therefore, some approaches try to generate the code directly from the model [7].

1.2 Research Question and Objectives

This thesis tries an approach from the other direction. First implementing the code, then trying to create the model and verifying it. The central problem of this approach is, again, the correlation between the model and the code. Therefore, the central research question can be formulated as follows:

Is it possible to create a UPPAAL model from C code in a way that can be automated?

This thesis first aims to define and implement the three protocols in *SmartOS*. The definitions should not be taken from some outside source but are directly derived from the author's understanding of the protocols obtained during the lecture on Real-Time Operating Systems held by Professor Marcel Baunach at Graz University of Technology.

The implementation avoids outside influence by not looking at existing implementations in various RTOSs to arrive at a fresh implementation not verified before. The resulting C code of the new implementation shall be the baseline for the translation to a UPPAAL model.

The translation shall be done manually while trying to condense the steps done to rules that could be useful for an automatic translation. As the graphs of the UPPAAL automata can not be easily obtained from the source code, they shall be as simple and small as possible, leaving as much of the logic as possible to the called updates, which can be directly translated from the source code.

This thesis does not aim to verify the implemented protocols thoroughly. Even though the models were used to find and fix bugs in the implementation, there can be no guarantee made for the correctness of the code using the presented models.

It is not an objective of this thesis to try to automate the translation process. It nearly aims to find out if such an automation could be feasible.

1.3 Thesis Structure and Organisation

After introducing and motivating the research question and objectives in Chapter 1, the following Chapter 2 aims to give more background information. This chapter will introduce UPPAAL, the terminology used in its models and give more background information about *SmartOS* and the terms used in the context of this RTOS.

A short section about the landscape of resource management in RTOSs is given in Chapter 3. This chapter also includes an overview of other works done around *SmartOS* concerning modelling and verification.

Chapter 4 aims to define the three examined protocols with a single statement each, from which all properties needed for the implementation are derived.

Chapter 5 is the heart of the thesis. The first section of this chapter gives an overview of how the protocols were implemented in *SmartOS* and how they can be used for an application. The second section concerns itself with describing the templates used for the UPPAAL automata, the central part of the modelling. The third, and final, section describes the rules found during the translation process, giving an example wherever possible.

The thesis ends with Chapter 6, where a conclusion is drawn and an answer to the research question is formulated.

CHAPTER 2

Background and Terminology

This chapter aims to give an overview of *SmartOS* and UPPAAL and explain the central terms that are used in their respective context.

2.1 SmartOS

SmartOS is a Real-Time Operating System (RTOS) developed and maintained in the Embedded Automotive Systems (EAS) group, led by Professor Marcel Baunach. Its primary purpose is to give the working group a flexible platform to conduct research where the base software is maintained in-house. In this section, only the most relevant elements for this thesis of *SmartOS* are highlighted. For a comprehensive description of the architecture and its research goals, see Scheipel et al. [8]. Whenever the text references a *SmartOS* Task, Event or Resource, it is indicated by capitalizing the word.

2.1.1 Tasks

The primary code execution containers of *SmartOS* are called Tasks, each consisting of a Task Control Block (TCB), code, and stack. The TCB contains information about the Task's base and current priority, information about the Task's current status, the register content of the Task's context if the Task is suspended, as well as meta-information about the resource management and the queues the Task is currently part of. Each Task has its code that gets executed if it is scheduled. The code execution of the Task starts in its entry function. Each Task has its stack that gets statically assigned at compile time. A Task in *SmartOS* can, at most, be part of two queues, one sorted by priority and one sorted by the next time-out. The priority queue can

either be the ready queue or an Event-specific queue, where the Tasks wait for the Event to be set. *SmartOS* features only one global timeout queue, where all Tasks are added that currently wait with a deadline for an Event or Resource, or are sleeping.

2.1.2 Events

Events are used to enable Tasks to communicate with each other. Events can be set and consumed by Tasks. If a Task wants to consume an Event that is not set at the moment, it has to wait. This can be done indefinitely or with a relative or absolute timeout. If a Task has to wait for an Event, it suspends itself, and the core is free to execute other Tasks.

2.1.3 Resources

Resources are used in *SmartOS* to synchronize the usage of shared resources, mostly global variables, between Tasks. A Task can hold onto multiple Resources at a time, but each Resource can be held by only a single Task at a time. The concrete implementation of the Resources depends on the used resource management protocol, which are the topic of this thesis.

2.1.4 Scheduling

SmartOS is a single-core Operating System (OS), significantly reducing the complexity of its scheduler. The scheduler in *SmartOS* is priority-based, meaning it always schedules the Task with the highest current priority. For this, a queue, called the ready queue, sorted by the Tasks' current priority is used. Tasks can decide to go to sleep or wait for an Event or Resource with a time-out. In all three cases, the Tasks get added to the global time-out queue and sorted by the earliest deadline first. Whenever a deadline is reached, the scheduler gets called and moves the timed-out Tasks from the time-out queue back into the ready queue. A Task in *SmartOS* can only ever be in one of three states: running, ready or waiting. If it is running, it is the head of the ready queue and is currently executed. Only one Task can be in this state at a time. If a Task is in the ready state, it is part of the ready queue but not the head, as it is not being executed. In the waiting state, a Task waits for either an Event or Resource or has put itself to sleep. In this state, a Task is part of the timeout queue if it waits for a deadline. Additionally, it is part of a priority queue if it waits for an Event or Resource.

2.2 UPPAAL

UPPAAL defines itself as an "integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays, etc.)." [9].

The UPPAAL version used in this thesis is UPPAAL 4.1.26-1. As not all UPPAAL functionalities are used in this thesis, only the relevant parts are described. For complete documentation, see the UPPAAL Documentation [10].

2.2.1 Automata

The basis of a UPPAAL system are the so-called automata. Each automaton is created from a template, a directed graph of locations and edges.

Locations

Four types of locations exist in UPPAAL. Initial locations, urgent locations, committed locations and regular locations.

Regular locations are the most basic form of location. In these locations, time is allowed to pass, i.e., the automaton may stay in this location while arbitrary transitions are happening in other automata.

Each template has to feature exactly one initial location, which serves as the starting location of the template's automata. The *initial location* is marked with a double circle, as seen in Figure 2.1. The initial location can also be urgent or committed.

While an automaton is at an *urgent location*, time is not allowed to pass, i.e., no transitions from regular locations are allowed in other automata. They can be identified by the "U" shape, as seen in Figure 2.1.

The last type of locations are the *committed locations*, marked by a "C", as seen in Figure 2.1. Committed locations are similar to urgent locations in that time is frozen while an automaton is at that location. In addition to the urgent locations, committed locations disable all transitions from locations that are not committed.

Edges

Edges mark possible transitions between locations. Edges can have guards, updates, synchronizations, selections and comments. Edges are directed, with the direction of the possible transition marked by the direction in which the arrow is pointing.

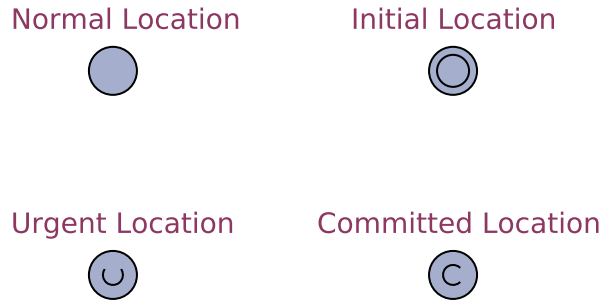


Figure 2.1: The different types of locations in UPPAAL.

Guards, depicted in green, only allow the transition if the expression evaluates to **true**. An example can be seen in Figure 2.2, where all the different annotations are depicted.

Updates, shown in blue in Figure 2.2, are used to evaluate expressions. In the scope of this thesis, updates are always used to call functions.

Synchronizations are used to enable two automata to interact with each other. Each synchronization consists of at least two annotations in different automata, marked with the same channel name. In the scope of this thesis, exactly one triggering edge and one triggered edge for each channel exist. When the triggering edge (marked with the **!** symbol) is transitioned, the triggered edge (marked with the **?**) is taken simultaneously, as long as the triggered edge is enabled. In this thesis, an automaton is said to be signalling another automaton via a channel when transitioning a triggering edge, leading to the transitioning of a triggered edge in the other automaton. Synchronizations are depicted in cyan, as seen in Figure 2.2. The models used in this thesis often use arrays of channels that can be indexed as normal arrays.

Selections bind an identifier, used in other annotations, to a value of the allowed range of a type. The example in Figure 2.2 shows the select annotation in yellow. In the models of this thesis, selections are always used to select an ID of a data structure, used to index the arrays in which the structures are saved. Selections allow the simulation to pursue different directions, as the value chosen is only limited by the range of the given type. Of course, the edge might be disabled for some values because of guards blocking the transition.

Comments are used to provide edges with notes to allow for a better understanding of the model. They do not have any effect. They are depicted in grey boxes, as seen in Figure 2.2.

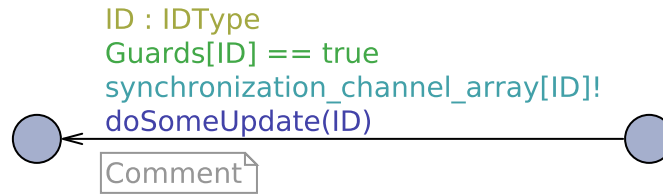


Figure 2.2: The different annotations of an edge in UPPAAL.

2.2.2 Parameters

Each template can optionally feature one or more parameters. Those parameters are used during system initialization to create the automata from the template. One automaton is instantiated for each possible value in the range of the parameter's type. The parameter's value acts as a constant variable in the scope of the corresponding automaton. In the scope of this thesis, parameters were used to instantiate multiple Task automata from the **Task** template, featuring the Tasks ID as the parameter.

2.2.3 Declarations

UPPAAL allows for C-like declarations of variables, structs, types, channels and functions. Some unique features are highlighted here.

Type definitions can feature special ranges if they are derived from the integer base type, as can be seen in Listing 2.1. This is a valuable feature as it leads to errors when a value outside the valid range is assigned to a variable of that type during simulation.

```
typedef int [lowerBound, upperBound] Type_t;
```

Listing 2.1: Definition of the integer type **Type_t** with the inclusive lower bound **lowerBound** and the inclusive upper bound **upperBound**. The two variables used for the bounds must be of the type **const int**.

UPPAAL features the boolean type **bool**, missing in C. As this type explicitly exists, using integer values as boolean expressions, often done in **if** statements in C, is not allowed.

UPPAAL does not allow for pointers, resulting in vastly different-looking code compared to the pointer-heavy *SmartOS* C code.

It is impossible to use forward declarations of functions in UPPAAL, leading to a reordering of the function declarations compared to the *SmartOS* source files.

CHAPTER 3

Related Work

Even though the decision was not to look at other implementations and definitions of resource management protocols, it is nevertheless interesting to look at which protocols are used by wildly adapted RTOSs.

In the automotive industry, AUTOSAR [2] and before that, OSEK [1] dominate the landscape. They both use statically defined Tasks that can be subdivided into Basic and Extended Tasks [11][12]. The former can not be preempted and always runs to completion, while the latter has the additional ability to enter a waiting state. As AUTOSAR has inherited the Task system from OSEK [12], they both use the OSEK priority ceiling protocol [11]. Even though the name suggests differently, this implementation follows the same concept as the Highest Locker Protocol (HLP) of this thesis instead of the Priority Ceiling Protocol (PCP) introduced in this work.

The European Space Agency (ESA) provides a prequalified version [13] of RTEMS [4], making it the de-facto standard RTOS in the European civil aerospace sector. RTEMS allows the user to choose between four different, so-called "Locking Protocols" [14], the Immediate Ceiling Priority Protocol (ICPP), the Priority Inheritance Protocol (PIP), the Multiprocessor Resource Sharing Protocol (MrsP), and the O(m) Independence-Preserving Protocol (OMIP). The ICPP is equivalent to the HLP and the PIP of RTEMS is equivalent to the PIP discussed in this thesis. MrsP and OMIP are generalizations for multi-scheduler and multi-processor systems and are not discussed further, as *SmartOS* is strictly a single-core OS and, therefore, all investigated protocols are for a single execution unit.

FreeRTOS [15] is probably the widest spread open-source RTOS currently available, defining itself as "a market-leading real-time operating system" [15]. Some prominent vendors like STMicroelectronics [16] even provide extensions for their Integrated Development Environ-

ments (IDEs), providing full integration of FreeRTOS into their software ecosystem [17]. The standard version of FreeRTOS only offers the PIP for its mutexes [18].

Zephyr [19] is a newer open-source RTOS with some of the largest global organizations, like Google and Intel, as partners and is part of the Linux Foundation. Like FreeRTOS, Zephyr uses the PIP as its resource management protocol [20].

As *SmartOS* [8] is a RTOS primarily used for research, quite some work has been done using it as a starting point. The works range from modular software updates [21], over OS-supported runtime-reconfiguration of the processor [22] to security extensions [23]. There has also been some work using models and code generation to create low-level source code for context switches in *SmartOS* [7]. The most closely related work has been done concerning modelling the kernel interface of *SmartOS* in UPPAAL [24]. Even though the UPPAAL models could not be directly used for this thesis, they have been used as an inspiration. *SmartOS* used a form of the HLP before the implementation was rewritten during the work done on this thesis and now offers all three protocols discussed in this work.

The modelling tool UPPAAL has been used for many different fields, from the formal design and analysis of a gear controller [25] over the verification of a planer for a deep space mission [26] and the verification of control software of a moon lander [27] to the modelling of cash withdrawals [28]. In the context of RTOSs, it is commonly used within schedulability analysis [29][30][31] to verify that the system does not violate any deadlines. Substantial research exists regarding the verification of OS kernels, with the complete formal verification of the seL4 microkernel [32][33] being the most well-known. Linux has also many works done regarding formal verification, seeing proposed approaches to verify the kernel [34], the eBPF runtime within the kernel [35] and a Linux hypervisor [36].

The concept of converting source code to models is nothing new, as it was already described by Holzmann et al. [37] in 2001. Nevertheless, the verification and validation of safety-critical code are still primarily based on testing, like in the examples of the automotive and civil space domains with AUTOSAR acceptance tests [3] and the processes described in the ECSS [38][39], showing that the transformation of code to model has not reached a stage where it can replace acceptance testing. Generation in the other direction, generating the code from a model, is more adopted with commercial tools like MATLAB Simulink [40] being available to the industry and investigations done about the usage of these tools in critical systems [41].

CHAPTER 4

Definition of Protocols

The protocols examined in this thesis all belong to the same general family of preemptive, priority-based scheduling protocols for a single execution unit. They range from a very simple protocol – HLP to a very complex protocol – PCP, with the PIP being of medium complexity. This chapter will define the protocols and describe the properties of their implementation.

4.1 Common Features

All three protocols are priority-based, so they always schedule the Task with the highest current priority. This current priority might not be static and may differ from the base priority, which is defined statically.

The protocols are preemptive, meaning Tasks can be disrupted in their execution. This is necessary as the current priority of the Tasks might change dynamically, and Tasks with higher current priority might become available during the execution of another Task.

The scheduler for all protocols is, therefore, more or less identical, as it always schedules the Task with the highest current priority to be executed. The difference between the protocols lies in how these current priorities are calculated. All protocols need base priorities that function as the minimum current priority of each Task. The current priority of a Task may be increased when a Task acquires a Resource or another Task wants to acquire a Resource held by the Task. The current priority might be decreased again when a Task releases a Resource or another Task stops waiting for a Resource held by the Task.

All protocols allow a Resource to be owned only by one Task simultaneously. It is possible for a Task to acquire a Resource multiple times. In this case, the Task also has to release the Resource the same amount of times it acquired it for the Resource to be free again.

4.2 Highest Locker Protocol (HLP)

The HLP is the most simple one of the three implemented protocols. It is based on a simple premise:

No other Task that might need the Resource during its execution is allowed to execute while a Task owns the Resource. (4.1)

This leads to a few properties:

- Tasks need to register all Resources they might use during their execution.
- A Task is not allowed to acquire a Resource it did not register.
- When a Task acquires a Resource, all Tasks that also registered that Resource need to have a lower current priority than this Task until the Task releases the Resource.

There are multiple ways to implement these properties. The version considered in this thesis adheres to the following additional properties:

- Each Resource has a priority that is the maximum of all the base priorities of the Tasks that registered it. This is called the ceiling priority of the Resource.
- This priority is static. Therefore, no (Resource-acquiring) Tasks can be added at run-time, and the base priority of (Resource-acquiring) Tasks can not be changed during run-time.
- When a Task acquires a Resource, its current priority gets set to the maximum of its old current priority and the ceiling priority of the Resource.
- A Task that owns Resources and has equal current priority as another Task which does not own any Resources gets scheduled first.
- Tasks that own a Resource are not allowed to sleep, as other Tasks that might want to acquire that Resource might be scheduled.

The base premise also leads to the protocol being free of deadlocks. To produce a deadlock, the system needs two Tasks to obtain two Resources in a specific order. Task 1 needs to obtain Resource A, then Task 2 needs to preempt Task 1 and obtain Resource B. Now Task 2 also needs to try to obtain Resource A. As Resource A is not free, Task 2 has to wait, and Task 1 is

scheduled again and is now trying to obtain Resource B. Now the system is in a deadlock state with Task 1 owning Resource A and waiting for Resource B and Task 2 owning Resource B, waiting for Resource A.

This chain of events is impossible in the HLP, as both Tasks must register both Resources. This leads to the current priority of Task 1 being raised to a high enough priority that it would not be preempted by Task 2 between obtaining Resources. Task 1 is, per definition, also not allowed to go to sleep at that point, so there is no way Task 2 can ever be executed and obtain Resource B, while Task 1 holds Resource A, preventing the deadlock situation.

4.3 Priority Ceiling Protocol (PCP)

The PCP is the most complex protocol of the three implemented ones. It is closely related to the HLP, with many common properties. Its base premise is also closely related to the one of the HLP in Definition 4.1, with the difference underlined:

No other Task that might need the Resource during its execution is allowed to get another Resource, while a Task owns the Resource. (4.2)

This change from *execute* to *get another Resource* prevents a higher priority Task from getting blocked when a lower priority Task holds a Resource, even though the higher priority Task would not need a Resource during its current execution.

The base premise again leads to a few properties:

- Tasks need to register all Resources they might use during their execution.
- A Task is not allowed to acquire a Resource it did not register.
- A Task is not allowed to acquire a Resource if there are Resources held by other Tasks that Tasks with equal or higher priority might acquire.

There are, again, multiple ways to implement these properties. The version considered in this thesis adheres to the following additional properties:

- Each Resource has a priority that is the maximum of all the base priorities of the Tasks that registered it. This is called the ceiling priority of the Resource.

- This priority is static. Therefore, no (Resource-acquiring) Tasks can be added at run-time, and the base priority of (Resource-acquiring) Tasks can not be changed during run-time.
- Each Task has a ceiling priority, which is the maximum of all the ceiling priorities of the Resources currently owned by the Task.
- There exists a ceiling priority for the complete system, which is the maximum of all the ceiling priorities of the Tasks currently owning Resources. This is called the system ceiling priority.
- A Task can acquire a Resource if the system ceiling priority is lower than its current priority.
- A Task can acquire a Resource if the system ceiling priority is higher or equal to its current priority only if the ceiling priority of all other Tasks is less than its current priority.
- If a Task can not get a Resource, it can wait until other Tasks release the Resources responsible for the blocking. The current priorities of the Tasks owning those Resources get raised to the maximum of their current priority and the current priority of the blocked Task.
- A Task can decide to wait for a Resource for a limited time. If the Task times out, the priorities of the Tasks, owning the blocking Resources, need to be recalculated.
- If a Resource is released and the system ceiling is lowered below the current priority of a Task waiting for a Resource, it is woken up and granted its Resource.

The base premise also leads to the protocol being free of deadlocks. The explanation follows the same example as in Section 4.2. After Task 1 has obtained Resource A, the system ceiling is raised to a level that is too high for Task 2 to obtain Resource B. Therefore, Task 2 has to wait for Task 1 to obtain Resource B and release both Resources again before being allowed to obtain the Resources and continue execution.

4.4 Priority Inheritance Protocol (PIP)

The PIP is different compared to the other protocols, as it never prevents a Task from getting a Resource if the Resource is free. Its base premise is:

If no Task currently owns a Resource, the Resource might be obtained by any Task.
If a Task blocks another Task, it will inherit the blocked Task's priority. (4.3)

This is the most straightforward protocol, as it eliminates the need of the other protocols to register Resource-Task pairs. The base promise of the PIP leads to the following properties:

- If a Resource is not owned by a Task, it can be obtained by any Task.
- The inheritance of the priority can be recursive.
- When a Task releases a Resource, the Task with the highest priority waiting for the Resource will get it.
- There might be chain-blocking where a Task can be blocked by a Resource that is held by another Task, which is also blocked by a Resource.

The implementation in this thesis adheres to the following additional properties:

- Each Resource has a priority, which is the highest priority of all the Tasks blocked by it.
- Each Resource has its own queue of Tasks waiting for it, which is sorted by the current priority of the Tasks.
- If a Task releases a Resource and another Task blocked by the Resource gets resumed, the Task's current priority needs to be recalculated.
- A Task can decide to wait for a Resource for a limited time. If the Task times out, the priority of the Resource might change, which could lead to a recursive recalculating of the whole blocking chain.

The protocol does not avoid deadlocks. This is easily shown with the same example used in Sections 4.2 and 4.3. After Task 1 has obtained Resource A, Task 2 can preempt it and obtain Resource B. When Task 2 tries to obtain Resource A, the priority of Task 1 gets raised, and Task 1 is executed again, while Task 2 waits for Resource A. When Task 1 tries to obtain Resource B, the System enters a deadlock.

4.5 Example

This example uses the same Task system for all three protocols to show their differences. It also shows a case where the HLP and the PCP avoid a deadlock situation while PIP gets deadlocked. The system consists of three different Tasks with different base priorities: Task L with the lowest priority, Task M with medium priority and Task H with the highest priority. The timelines of each Tasks execution, if no other Tasks would be present, can be seen in Tables 4.1, 4.2 and 4.3. It is important to note that Tasks M and H only wake up after a certain time.

Action	Duration	Time
Wake Up		0
Execute	1	
Get Resource R1		
Execute	3	
Get Resource R2		
Execute	1	
Release Resource R2		
Execute	1	
Release Resource R1		
Execute	1	
Terminate		

Table 4.1: The timeline of Task L.

Action	Duration	Time
Wake Up		2
Execute	2	
Terminate		

Table 4.2: The timeline of Task M.

Action	Duration	Time
Wake Up		3
Execute	2	
Get Resource R2		
Execute	1	
Get Resource R1		
Execute	1	
Release Resource R1		
Execute	1	
Release Resource R2		
Execute	1	
Terminate		

Table 4.3: The timeline of Task H.

The timelines are again shown in Figure 4.1, where the Tasks execute in a true parallel fashion as if the system has three execution units. The timeline when executing using the HLP can be seen in Figure 4.2, the timeline when using the PCP in Figure 4.3 and the timeline when using the PIP can be seen in Figure 4.4. Task L is shown in green, with the Resource actions shown below the nodes, starting at the lowest priority. Task M is shown in blue, with medium priority. Task H is drawn in red, starting last and with the highest priority. Priority actions related to Task H are labelled above their nodes. If Resource x gets acquired, it is shown as $+R_x$. The node is labelled as $-R_x$ if Resource x gets released. If a Task tries to acquire Resource x , but the Resource is not free, the Task gets blocked as a result, and the node is labelled as $(+R_x)$. Whenever a Task is ready but not executed, its corresponding line is dashed. The deadlock scenario described in Section 4.4 can be observed in Figure 4.4.

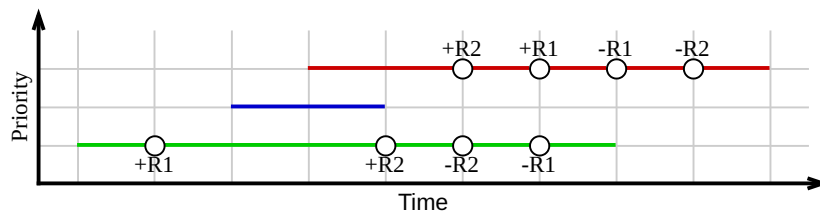


Figure 4.1: Parallel timeline of the example, where all Tasks get executed simultaneously.

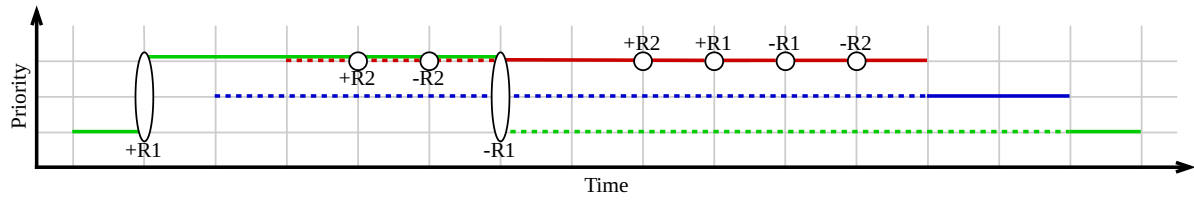


Figure 4.2: The timeline using the HLP, showing the priority changes during the execution of the three Tasks.

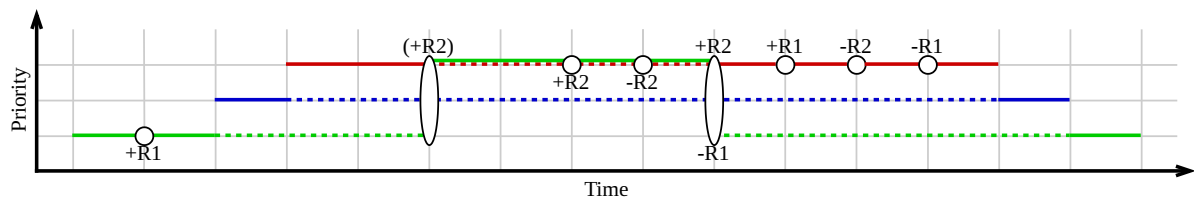


Figure 4.3: The timeline using the PCP, showing the priority changes during the execution of the three Tasks.

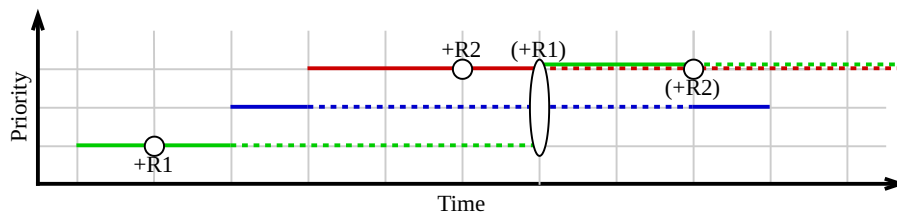


Figure 4.4: The timeline using the PIP, showing a deadlock scenario where both Task L, in green, and Task H, in red, are blocked forever.

CHAPTER 5

From Code to Model

5.1 Code Implementation

After the protocols were defined, they were implemented in the context of *SmartOS*. These code implementations were written in plain C [42] using the existing *SmartOS* environment. A separate directory for each protocol was created to allow easy switching between the protocols, where all protocol-specific code resides. The protocol can be chosen in the **Makefile** of the application by setting the **RESOURCE_MANAGEMENT** variable, which only builds the directory of the chosen protocol. A key objective of the implementation was to maximize the amount of shared code. This led to convoluted structures inside the kernel and some very compiler-specific language usage of the GCC compiler.

5.1.1 Directory Structure

The relevant directories and files can be seen in Figure 5.1. The only places where protocol-specific code can be found are the respective directories in the **src/kernel/** directory. The protocol-specific data structures are defined in the **os_resource_types.h** files. The correct file gets included in the rest of the OS via the **os_data_structures.h** file, which includes all other data structures in the **include/kernel/kernel_data_structures** directory. All syscall wrappers are protocol agnostic and can be found in the **kernel.h** header file, with the protocol-specific implementation in the **.c**-file in the respective protocol-specific directory in **src/kernel**. As the data structure for the Tasks also needs to contain protocol-specific information, the Task data struct contains a struct that contains all this protocol-specific information. This protocol-specific struct called **TaskResourceManagementStruct_t** is

defined differently for each protocol and defined in the respective `os_resource_types.h` file.

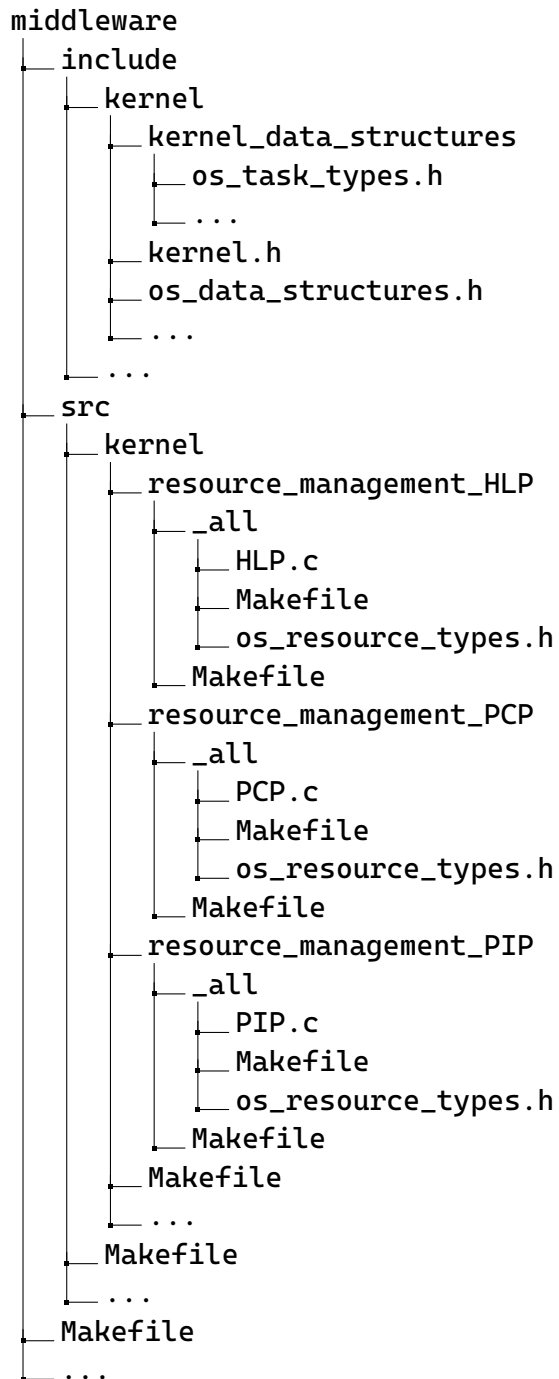


Figure 5.1: The directory tree of the relevant directories and files for the resource management protocols.

5.1.2 Including the Chosen Resource Management Protocol

As the path of the files containing the definition of the data structures needed for each resource protocol changes depending on the `RESOURCE_MANAGEMENT` variable in the `Makefile`, the include paths need to be variable as well. This is done with the two lines seen in Listing 5.1 of the `os_data_structures.h` file. The first line of the listing is very compiler-specific, as the `RESOURCE_MANAGEMENT` variable that gets set from the `Makefile` needs to be replaced by the preprocessor. The space in front of the `RESOURCE_MANAGEMENT` gets deleted in the process, resulting in a valid file path. This deletion is only done using GCC [43], not with Clang [44], therefore restricting *SmartOS* to GCC, as the file path is not allowed to contain spaces. Removing the space before the `RESOURCE_MANAGEMENT` identifier leads to the compiler no longer registering it as a variable, leading to an invalid file path not specifying the protocol. The compiler used for *SmartOS* has always been GCC. Therefore, the restriction to GCC does not outweigh the cleanness of the directory tree.

```
#define INC_RESMNG      <../src/kernel/resource_management_  
    RESOURCE_MANAGEMENT/_all/os_resource_types.h>  
#include INC_RESMNG
```

Listing 5.1: Code used to include the `os_resource_types.h` file.

5.1.3 Resource Management Interface

Each resource management protocol needs to implement the functions, structs and macros needed for the common interface. Some of the functions or macros might not be needed for each protocol. It was decided to require each protocol to implement everything to allow for application code compatibility. The function headers of the syscall wrappers are shown in Listings 5.2, 5.3, and 5.4. The headers of the syscalls can be seen in Listings 5.5 and 5.6. Some additional functions are needed inside the kernel, with their headers in Listings 5.7, 5.8, 5.9, and 5.10. The names of the needed structs can be found in Listings 5.11 and 5.12. The macros can be seen in Listings 5.13 and 5.14.

```
Result_t getResourceUntil(Resource_t* Resource, Time_t* Deadline);
```

Listing 5.2: Tries to get a Resource. If the Resource is unavailable, the Task waits until the absolute deadline **Deadline** is reached. For HLP the deadline is ignored, as the Task will always get the Resource without waiting

```
Result_t getResource(Resource_t* Resource);
```

Listing 5.3: Same functionality as `getResourceUntil` but the Task will wait indefinitely for the Resource if it is not freed.

```
Result_t releaseResource (Resource_t* Resource);
```

Listing 5.4: The Task releases the Resource.

```
void syscall_getResourceUntil(Resource_t* Resource, Time_t* Deadline);
```

Listing 5.5: The syscall for getting a Resource with a deadline (deadline ignored for HLP).

```
void syscall_releaseResource(Resource_t* Resource);
```

Listing 5.6: The syscall for releasing a Resource.

```
Result_t os_isAllowedToSuspend(Task_t* Task);
```

Listing 5.7: A check done when a Task tries to go to sleep. Important for HLP as Tasks are not allowed to sleep while holding a Resource. For PCP and PIP this function always returns **SUCCESS** as the protocols allow for Tasks to sleep at any time.

```
void os_handleRequestExpired(Task_t* Task);
```

Listing 5.8: A handler called when a Task, which was waiting for an Event, timed out. The Event might have belonged to the resource management protocol and needs to be handled differently depending on the protocol used. For HLP, it simply removes the Task from the queue waiting for the Event, as HLP does not need any Events.

```
void os_initTaskResourceManagement(TaskResourceManagementStruct_t*
    ResourceManagementStruct);
```

Listing 5.9: Initializes the protocol-specific data structure for Tasks. This has to be done during the startup procedure, as the TCBs only gets created at that time.

```
void registerResources();
```

Listing 5.10: This function gets called during startup. HLP and PCP calculate the needed ceiling priorities during that call. PIP does not need to do anything here, as no ceiling priorities are needed.

```
Resource_t
```

Listing 5.11: The central data structure that needs to contain all the data of a Resource, needed for the syscalls.

```
TaskResourceManagementStruct_t
```

Listing 5.12: Data structure that is part of the TCB and contains all the Task-specific data needed for the protocol.

```
OS_DECLARE_RESOURCE(_name)
```

Listing 5.13: Macro to create a new Resource. Allocates and initializes the needed data structures.

```
OS_REGISTER_RESOURCE(_taskName, _resourceName)
```

Listing 5.14: Macro to register a Task Resource pair.

5.2 Model

Each system, one for each protocol, features two kinds of automata templates. One represents the control flow directly associated with a Tasks execution flow. The other template represents everything that happens during the switch between Tasks, like removing the Task from its current queue and calling the scheduler. The first automata template is called **Task**, while the other is called **Kernel**. It is important to note that the naming of the templates does not correspond to the mode the OS would be in during the execution of code corresponding to the template. For example, most of the logic of the syscalls is handled in the **Task** automata, only switching to the **Kernel** automata when the scheduler is needed or the syscall leads to switching Tasks. In contrast, the OS would be in kernel mode as soon as the Task enters the syscall.

5.2.1 Base Model

Before starting the modelling of the different protocols, a base model was created that did not feature anything related to resource management. The two templates can be seen in Figures 5.2 and 5.3.

The Task template features the three Task states as described in Scheipel et al. [8], in the form of the **Running**, **Ready** and **Waiting** locations. To ensure that not all Tasks can be in **Ready** location, while time is passing, the location is declared as urgent. The **Task** template has one parameter called **TaskID**. This parameter is of type **TaskIDLegal_t**. The system, therefore, has **NumberTask** + 1 instances of the **Task** automata. This is done in order to always have a

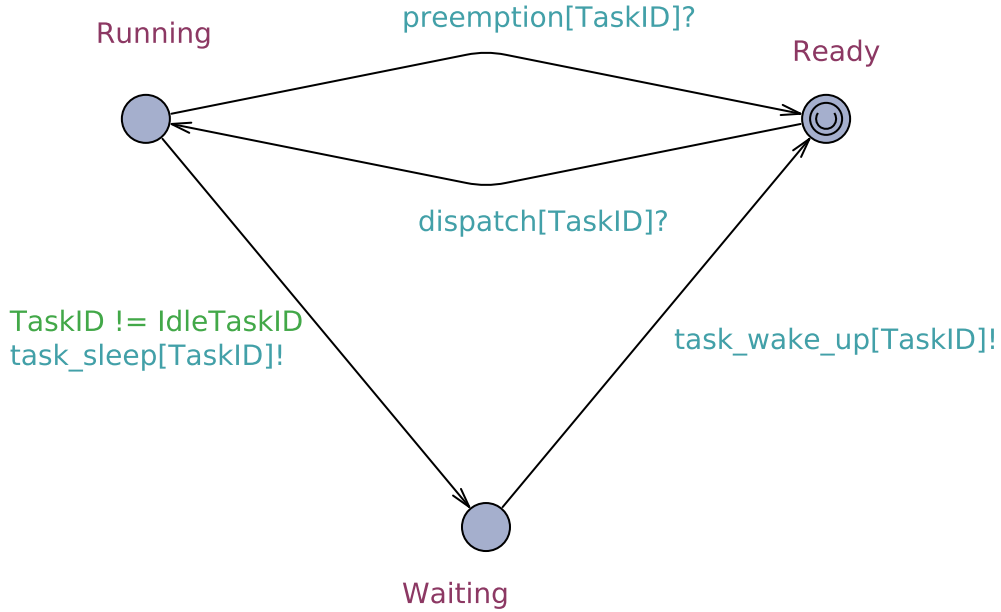


Figure 5.2: Template for the **Task** automata of the base model. The template has one parameter: `const TaskIDLegal_t TaskID`.

dedicated *Idle Task* that is not allowed to enter the **Waiting** location. This Task is defined to have ID `IdleTaskID = 0`.

A Task immediately enters the **Ready** location, which is consistent with *SmartOS*. A Task will stay in this location until it gets the signal to move to the **Running** location via the **dispatch** channel with its ID. While the Task is in the **Running** location, time is allowed to pass until the Task either gets preempted via a signal on the **preemption** channel with its ID, going back to the **Ready** location or the Task goes to sleep changing to the **Waiting** location and sending a signal via its **task_sleep** synchronization channel. The latter transition is guarded by a check if the Task is the *Idle Task* to ensure that this Task is never in the **Waiting** location. While a Task is in the **Waiting** location, time is allowed to pass until the Tasks wakes up, firing a signal in its **task_wake_up** channel and switching to the **Ready** location.

This model of a *SmartOS* Task abstracts nearly everything and reduces the automata to an absolute minimum. Even Events, a central element of *SmartOS*, can be abstracted away. Setting an Event does not change anything for the Task that sets it apart from the Task being preempted by a higher priority Task, which is modelled by the edge from **Running** to **Ready** location. When a Task wants to wait for an Event, and the Event was already set, it just continues in the **Running** location. In *SmartOS*, the Task would, of course, use a syscall and, therefore, needs to switch to kernel-mode, but the model in this thesis is not concerned with

the current mode of the OS. If the Task waits for an Event that is not set, it will change to the **Waiting** location. It is not important for this model which other Task will set the Event or if the Task will time-out waiting for it, as the result from a Task perspective is the same. It will change to **Ready** location after some time has passed in the **Waiting** location. Another way of seeing this is that waiting for an Event is nothing but sleeping for a time determined by another Task. As this model is not concerned with interactions between Tasks, Events can be abstracted away as a form of sleeping. This is a key difference to the models used for *SmartOS* in other publications [24][45], as those publications looked more into modelling the interface between the kernel and user space and interactions between the Tasks. In contrast, this thesis looks into modelling parts of the kernel, with the Tasks only being the trigger of kernel functionality.

The template for the kernel automaton, seen in Figure 5.3, features no parameters and, therefore, only has one instance in the system. All its locations are committed or urgent, as time can only pass while a Task is being executed. The kernel automaton starts out in the **InitialState** location. The `initKernel` update is called when leaving the initial location. This initializes all the needed data structures for Tasks, with the protocol-specific models also initializing everything that is needed for the resource management.

The kernel then enters the **Scheduler** location, which is committed, so it immediately switches to the **Dispatcher** while calling the `os_schedule` update. This update sets the variable `os_RunningTask` to the ID of the head of the ready queue, i.e., it ensures that the next Task that gets executed is the Task with the highest priority of all Tasks in the **Ready** location. After the **Dispatcher**, the kernel sends a signal on the `dispatch` synchronisation channel of the `os_RunningTask` while switching to the **TaskExecution** location. This allows the Task with ID `os_RunningTask` to switch from **Ready** to **Running**.

In the **TaskExecution** location, the kernel automaton waits until one of the Tasks sends a signal. Either one of the Tasks that are currently in the **Waiting** location sends a signal on its `task_wake_up` channel, which prompts the kernel automaton to add the Task back into the ready queue with the `os_insertPriority` update, before preempting the currently running Task and going back to the **Scheduler** location. This path needs an intermediate location called **Intermediate1**, as it is not possible to have two synchronizations in one location. The second path the kernel automaton can take to move from the **TaskExecution** location back to the **Scheduler** location involves the currently running Task to switch from **Running** to **Waiting** while sending a `task_sleep` signal. Here, the kernel removes the Tasks ID from the ready queue before returning to the **Scheduler** location.

The most striking difference between this model and the *SmartOS* implementation is that

no timeout queue is maintained. In *SmartOS*, this queue tracks the time-outs of Tasks while sleeping or waiting for an Event. As the timings are irrelevant to the model, the whole queue could be removed, significantly reducing the system's state space. Reducing the model's state space as much as possible is a key objective throughout the modelling process, as it is the main contributor to run time and memory usage during simulation and verification.

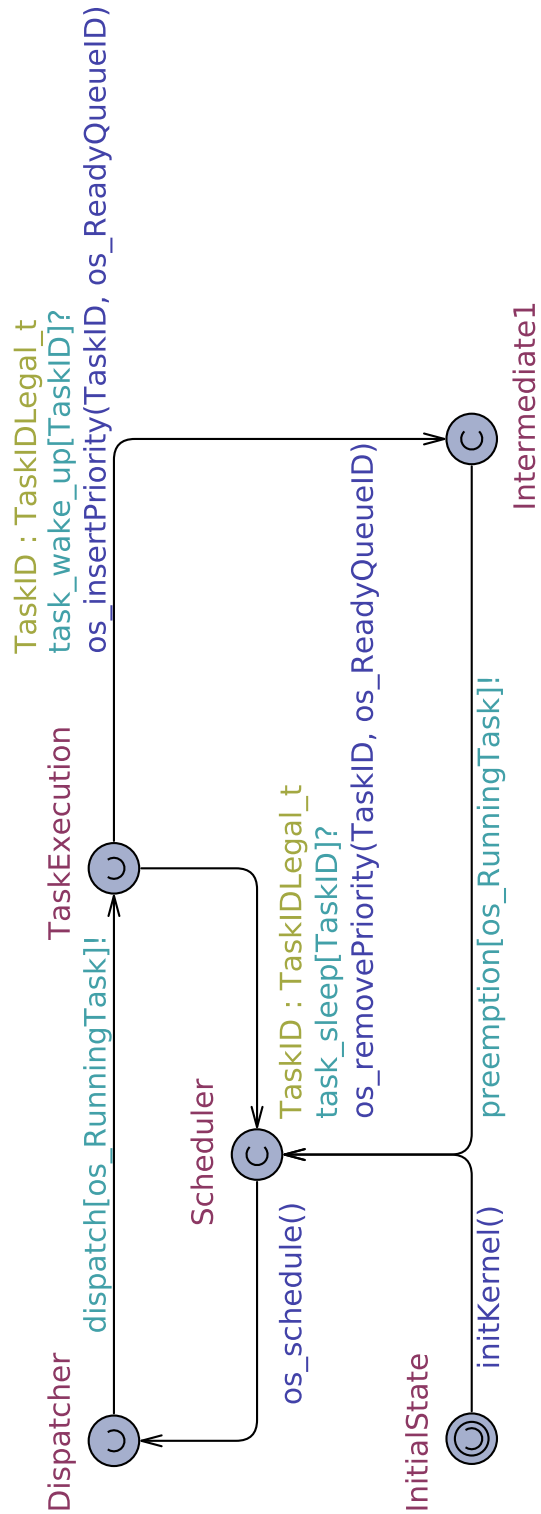


Figure 5.3: Template for the `Kernel` automata of the base model. The template has no parameters.

5.2.2 HLP

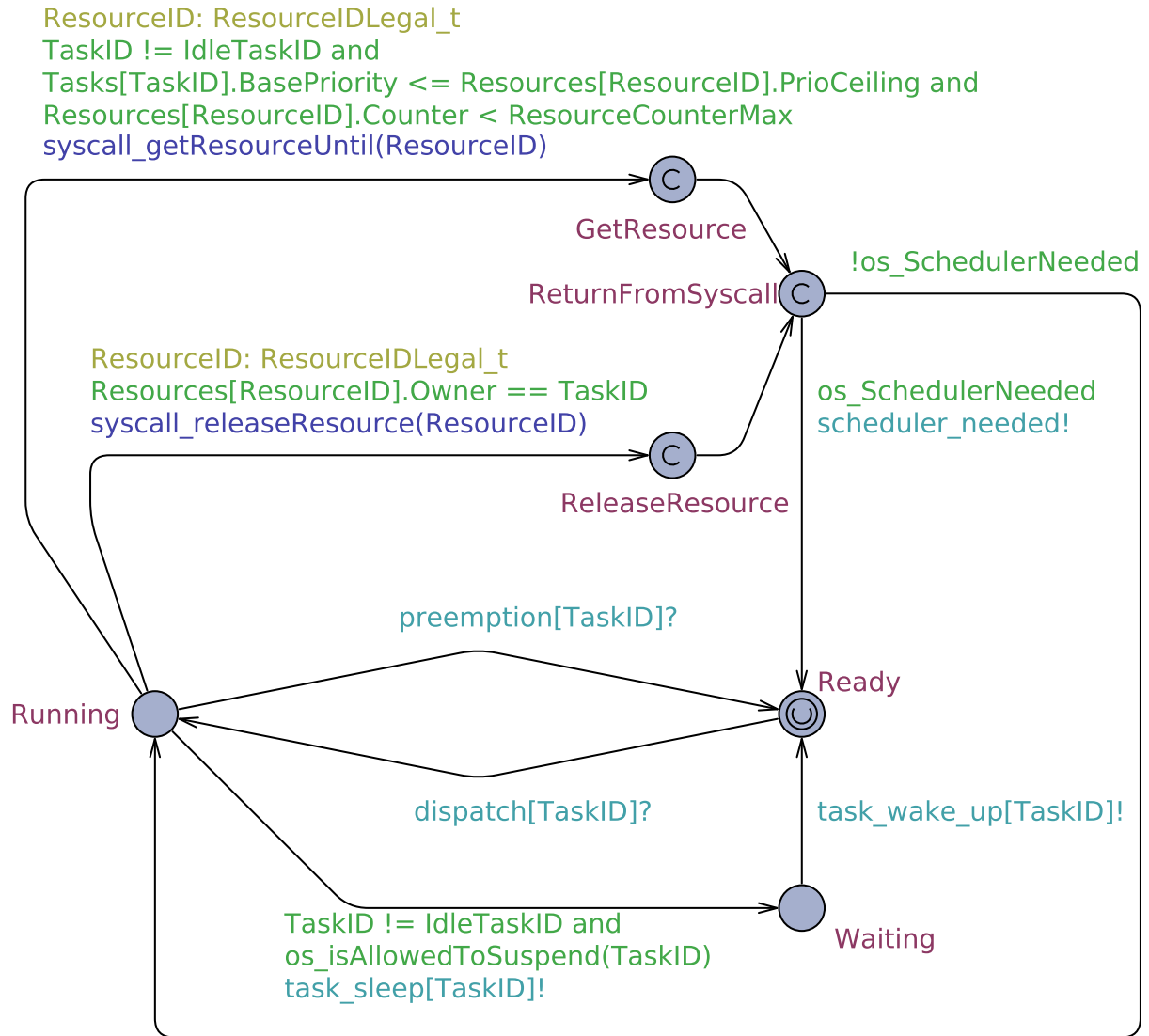


Figure 5.4: Template for the **Task** automata of the HLP model. The template has one parameter: `const TaskIDLegal_t TaskID`.

The template for the **Task** automata for the HLP protocol can be seen in Figure 5.4. It features the same three basic states featured in the template of the base model in Figure 5.2. The only difference in this basic triangle is that the guard between the **Running** and **Waiting** location has changed from only checking the Task ID to additionally calling one of the interface functions, described in Listing 5.7, checking if the Task can go to sleep.

Three additional locations have been added to model the behaviour of the resource management protocol. One location for obtaining a Resource, called **GetResource**, one location

for releasing a Resource, called **ReleaseResource**, and one location used for a unified return from the two syscalls, called **ReturnFromSyscall**.

The **GetResource** location can be reached from the **Running** location and calls the **syscall_getResourceUntil** update, which has the same functionality as the syscall described in Listing 5.5. The transition is guarded by multiple clauses to prevent a Task from obtaining a Resource it is not allowed to get. First of all, the *Idle Task* is never allowed to own a Resource. Secondly, a Task is only allowed to get Resources if the Task's base priority is lower or equal to the ceiling priority of the Resource. This simulates that Tasks must declare what Resources they might use during their execution using the **OS_REGISTER_RESOURCE** macro described in Listing 5.14. As the model aims to be as general as possible, no Task Resource pairs have to be declared in the system. All Resources with high enough ceiling priority can be obtained by all Tasks with low enough base priority. As the system verifies all possible states, the states where a Task never obtains a specific Resource are also considered. This represents the case where the Task would never use the Resource and, therefore, would not register the Task Resource pair. The third guard is necessary to make the state space finite. As defined in Section 4.1, a Task can own a Resource multiple times. As this is not bounded per definition, the state space would be infinite. To prevent this, a global constant called **ResourceCounterMax** is introduced, which prevents the Task from owning a single Resource more times than the constant specifies. The constant is set to 2, as owning a Resource more than twice does not open up new control flow paths.

The **ReleaseResource** location can be reached from the **Running** location and calls the **syscall_releaseResource** update, which has the same functionality as the syscall described in Listing 5.6. The guard ensures that only Resources that are actually owned by the Task can be released.

Both described locations are committed, meaning no transitions that do not leave the location can be taken. This ensures that no time passes in those locations. Both locations always transition to the **ReturnFromSyscall** location, which is again committed. This location acts as a unification point, as the following guards and transitions are independent of the previous path. The next transition depends on the **os_SchedulerNeeded** boolean variable, which can be set during the updates in the previously described transitions. If the variable is **false**, the head of the ready queue has been unchanged, and the Task can continue executing, transitioning to the **Running** location. If **os_SchedulerNeeded** is **true**, the head of the ready queue has changed, needing the **Kernel** automaton to start the process of dispatching a new Task, signalled via the **scheduler_needed** channel. In this case, the Task must go to the **Ready** location.

The template for the **Kernel** automaton can be seen in Figure 5.5. The template is the same as the one of the base model seen in Figure 5.3 except for a new transition between the **TaskExecution** and **Scheduler** locations waiting for a signal on the **scheduler_needed** channel.

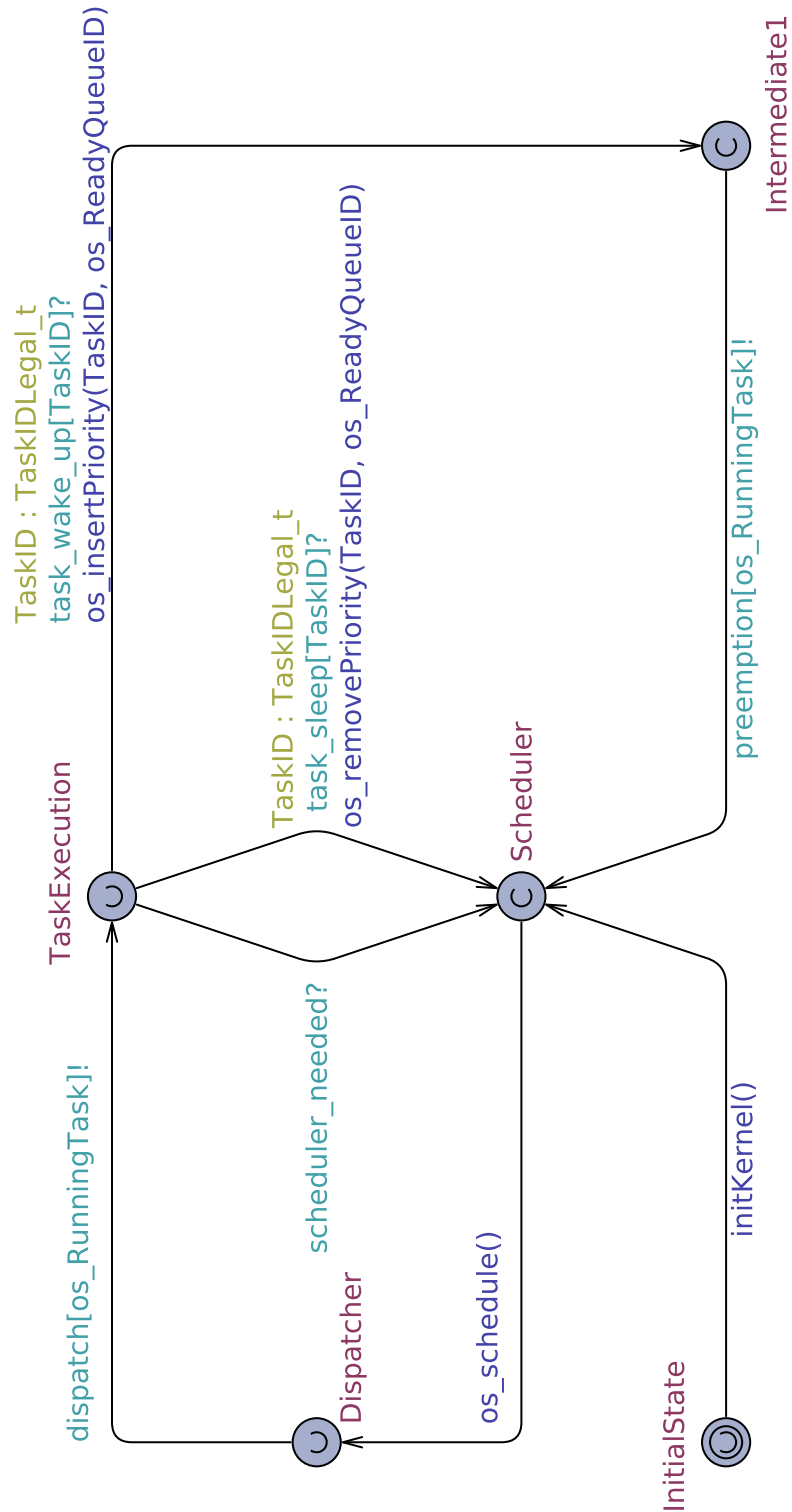


Figure 5.5: Template for the `Kernel` automaton of the HLP model. The template has no parameters.

5.2.3 PCP

The template for the **Task** automata for PCP, seen in Figures 5.6 and 5.7, only differs in a few places from the one for HLP seen in Figure 5.4. As it only adds some additional paths to the model, only those differences will be explained here. For the explanations of the shared parts, refer to Section 5.2.2.

Instead of a single direct path from the **GetResource** location to the **ReturnFromSyscall** location in the HLP model, PCP needs to account for the situation where the Resource is not granted, and the Task needs to wait. The split is done using the **PriorityListHeadID** saved in the Tasks TCB. If the Task is still part of the ready queue, the Resource was granted, and the Task can follow the same path as it would in the HLP model. If this is not the case, the Task signals the kernel that it needs to wait using the **task_wait_for_resource** channel and go to the **WaitingForResourceEvent** location. From this location, the Task can take two paths to get back to the **Ready** location. One simulates that the Task has timed out, prompting the Task to send a signal using its **task_timed_out** channel. If the Tasks **resource_event_set_task** channel gets signalled by another Task, during the time the Tasks spends waiting in the **WaitingForResourceEvent** location, the Task can also go back to the **Ready** location.

The other place where new paths have been added is between the **ReleaseResource** location and the **ReturnFromSyscall** location. The simple direct connection has been split into two paths, depending on the helper variable **helper_ResourceEventSet**. This variable gets set to **true** if the special **os_EventPCP** Event was set during the **syscall_releaseResource** update. If the Event was set, the channel **resource_event_set_task** channel of the corresponding Task is signalled, prompting that Task to wake up. The updates on this edge reset the helper variables to keep the state space smaller.

The template of the **Kernel** automaton of the PCP, seen in Figure 5.8, features two new edges compared to the one of the HLP, seen in Figure 5.5.

The first edge accounts for the new possibility of Tasks waiting for a Resource, removing them from the ready queue, leading to another Task being expected. For this, the **Kernel** automaton is switching from the **TaskExecution** location to the **Scheduler** location whenever a Task sends a signal on the **task_wait_for_resource** channel.

The second new edge goes from the **TaskExecution** location to the **Intermediate1** location, waiting for a Task to signal on its **task_timed_out** channel when timing out during a wait for a Resource. The **os_handleRequestExpired** update ensures that all priorities get updated correctly, while the **os_insertPriority** update inserts the Task back into the ready queue.

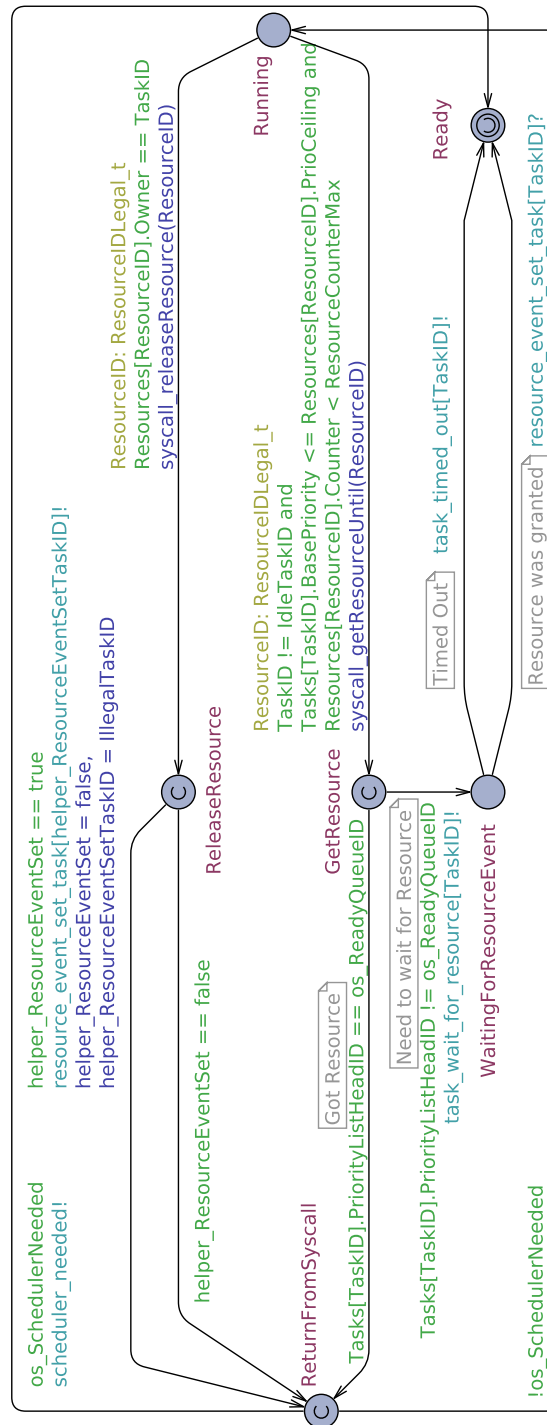


Figure 5.6: The first part of the template for the **Task** automata of the PCP model. The template has one parameter: `const TaskIDLegal_t TaskID`. The model misses edges between the **Ready** and **Running** Location as well as the **Waiting** location and all its edges shown in Figure 5.7.

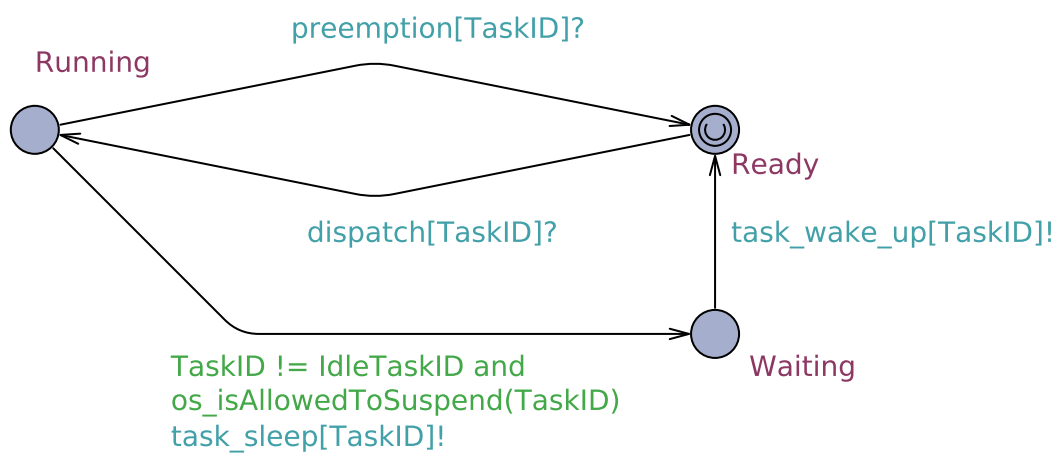


Figure 5.7: The second part of the template for the **Task** automata of the PCP model. This shows the remaining edges and locations of Figure 5.6

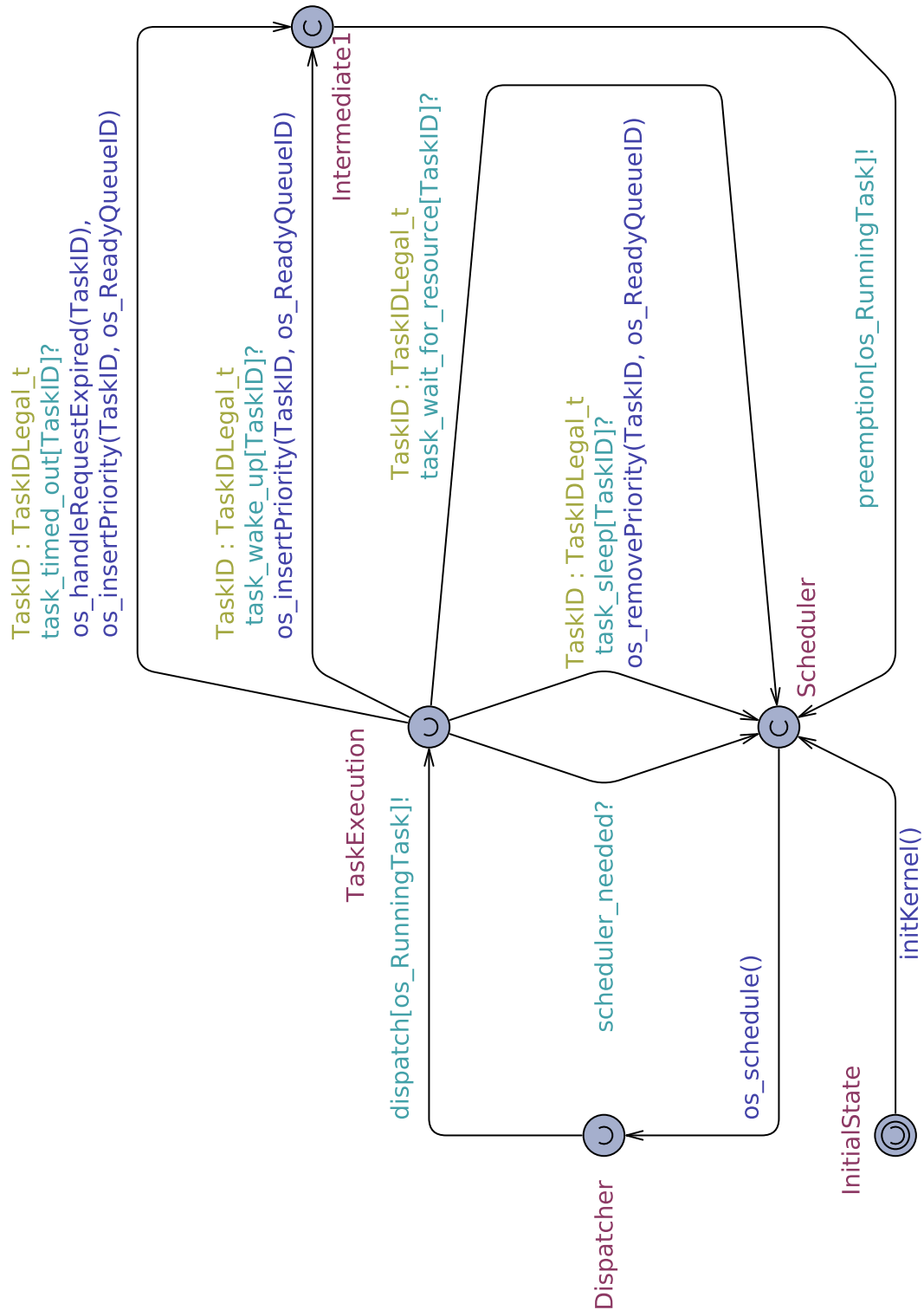


Figure 5.8: Template for the `Kernel` automaton of the PCP model. The template has no parameters.

5.2.4 PIP

The two templates for the PIP **Task** and **Kernel** automata, shown in Figures 5.9, 5.10 and 5.11 are identical to the templates for the PCP shown in figures 5.6, 5.7 and 5.8, with one exception. The **Task** template of the PIP in Figure 5.9 does not feature the guard expression `Tasks[TaskID].BasePriority <= Resources[ResourceID].PrioCeiling` between the **Running** location and the **GetResource** location because the Resources in the PIP do not feature priority ceilings.

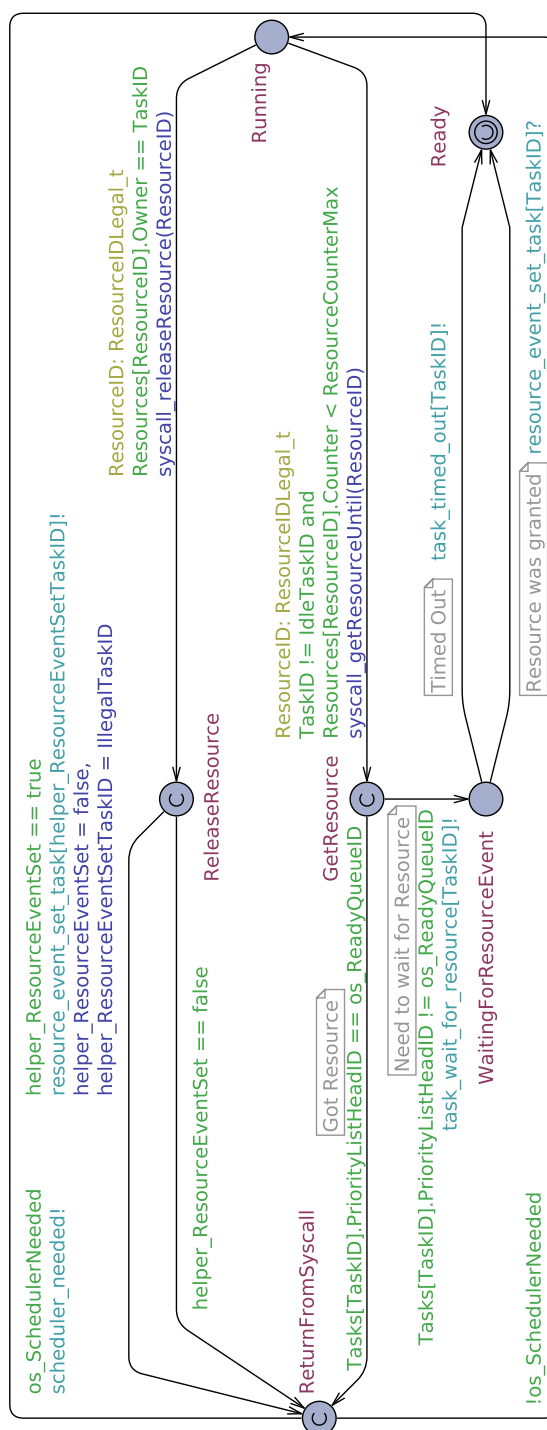


Figure 5.9: The first part of the template for the **Task** automata of the PIP model. The template has one parameter: `const TaskIDLegal_t TaskID`. The model misses edges between the **Ready** and **Running** Location as well as the **Waiting** location and all its edges shown in Figure 5.10.

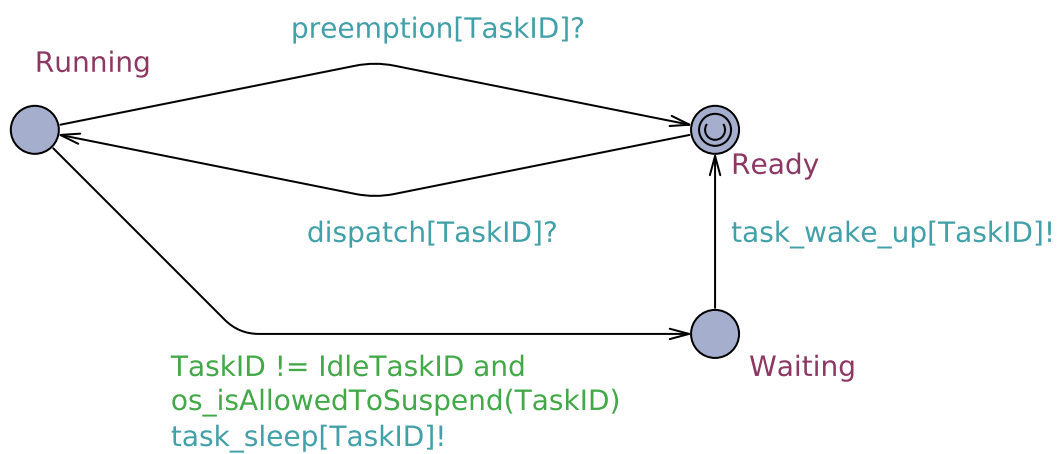


Figure 5.10: The second part of the template for the **Task** automata of the PIP model. This shows the remaining edges and locations of Figure 5.9

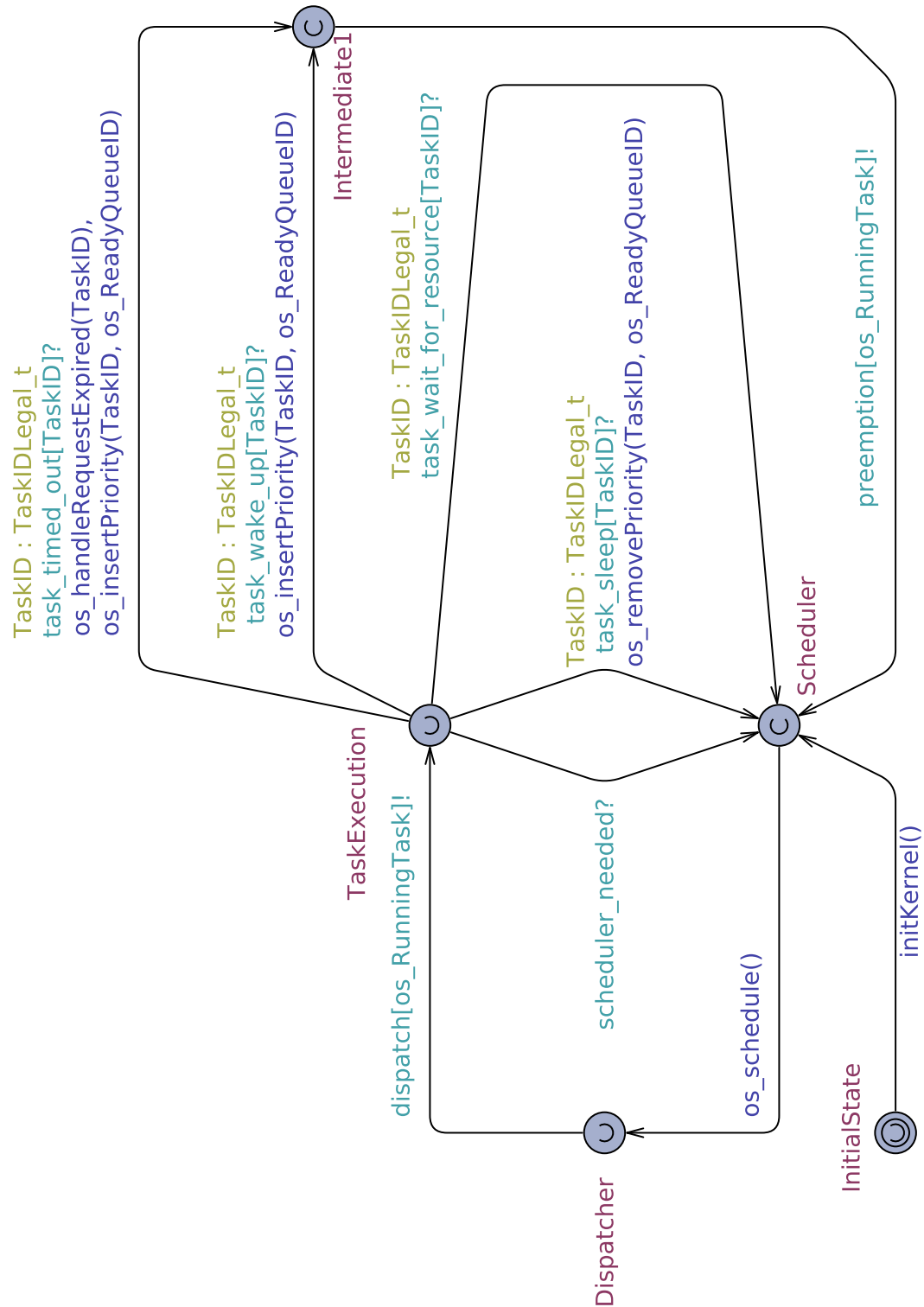


Figure 5.11: Template for the `Kernel` automaton of the PIP model. The template has no parameters.

5.3 Transforming Code

After all the manual work of expressing the execution flow to automata templates has been done as described in Section 5.2, the source code of *SmartOS* can be translated from C to code that is compatible with UPPAAL. As this thesis investigates a general concept, engineering a translator is out of scope. Therefore, the goal was to develop rules for the translation. As the rules are independent of the used protocol, all following examples are taken from the base model or the HLP implementation, as it is the most simple protocol. In this chapter's listings, a grey background indicates code in UPPAAL, while the beige background indicates *SmartOS* C source code.

5.3.1 Pointers

The most significant difference between C code and the C-like code UPPAAL uses is that UPPAAL does not support pointers. This means that everything handled with pointers in C must be done differently. In the scope of this thesis, it is possible to substitute pointers with arrays and IDs, which are used as indices for the arrays. Pointers often use the value `NULL` to indicate that the pointer is invalid. To achieve this functionality, all IDs that span the range of 0 to a maximum instead start from -1 . This addition of an "illegal" value for the IDs comes with the problem that UPPAAL often uses the range of a type for additional things, like instantiating a specific amount of automata from a template or using the whole range when evaluating a *Select* statement during a transition. The solution was to introduce two types in UPPAAL per pointer type, one with a range starting at 0 and one starting at -1 . The one starting from 0 has the postfix `Legal`, as seen in Listing 5.15, where all the different ID types are shown. The additional `TaskIDProper_t` type was introduced for `for`-loops over all Tasks, excluding the *Idle Task*.

```
typedef int [ 1, NumberTasks]      TaskIDProper_t;
typedef int [ 0, NumberTasks]      TaskIDLegal_t;
typedef int [-1, NumberTasks]      TaskID_t;
typedef int [ 0, NumberResources - 1] ResourceIDLegal_t;
typedef int [-1, NumberResources - 1] ResourceID_t;
typedef int [ 0, NumberPriorityLists - 1] PriorityListIDLegal_t;
typedef int [-1, NumberPriorityLists - 1] PriorityListID_t;
```

Listing 5.15: Different ID types used to replace pointers. Not all protocols use all ID types, e.g., HLP does not use **ResourceID_t**.

The upper bound of the ranges shown in Listing 5.15 are one less than a constant in most cases to account for the lowest ID being 0. The only exceptions to this rule are the IDs of Tasks, as the *Idle Task* has ID 0 and is not counted to the number of regular Tasks.

As Tasks, Resources and list heads exist freely in the memory space of the C implementation of *SmartOS* and can only be referenced using pointers, they have to be grouped into arrays in UPPAAL. The arrays can be seen in Listing 5.16. The array length for the Tasks is extended by one to account for the *Idle Task*.

```
Task_t      Tasks      [NumberTasks + 1];
Resource_t  Resources   [NumberResources];
TaskID_t    PriorityListHeads [NumberPriorityLists];
```

Listing 5.16: Arrays used in combination with IDs to replace pointers.

5.3.2 Types

Only one type is directly taken from *SmartOS* and is needed in the UPPAAL models. Its C definition can be seen in Listing 5.17, with its UPPAAL equivalent in Listing 5.18. The range for the UPPAAL type is bounded by the number of Tasks, as there can only be this many different priority levels.

```
typedef uint16_t Priority_t;
```

Listing 5.17: Definition of the `Priority_t` type in *SmartOS*.

```
typedef int[0,NumberTasks] Priority_t;
```

Listing 5.18: Definition of the `Priority_t` type in the UPPAAL models.

5.3.3 Data Structures

SmartOS features several data structures, with some of them needed for the UPPAAL model. Most data structures in *SmartOS* feature several pointers that had to be changed to IDs in UPPAAL. Additionally, everything regarding time-outs has been abstracted away, as described in Section 5.2.1.

The translation of the `ListElement_t` struct seen in Listings 5.19 and 5.20 is straightforward, as only the pointers to the Tasks need to be changed to IDs. This structure is used for the double-linked lists used for the different queues in *SmartOS*.

```
typedef struct {  
    struct Task_s* Prev;  
    struct Task_s* Next;  
}ListElement_t;
```

Listing 5.19: Definition of the `ListElement_t` struct type in *SmartOS*.

```
typedef struct {  
    TaskID_t Prev;  
    TaskID_t Next;  
} ListElement_t;
```

Listing 5.20: Definition of the `ListElement_t` struct type in UPPAAL.

Even though the UPPAAL models do not use Events in their pure form, as described in Section 5.2.1, they are still needed within the PCP and PIP protocols. The translation of the structs seen in Listings 5.21 and 5.22 is again straightforward, as the pointer to the list head is changed to an ID. The type of the `Value` field is changed from an integer type to a boolean, as it can only ever have the two values 0 and 1. It would have been possible to create an additional integer type in UPPAAL with the range 0 to 1, but as the value is always checked against being 0 or not, it was decided to change it to boolean.

```
typedef struct Event_s{
    Word_t      Value;
    struct Task_s* ListHead;
} Event_t;
```

Listing 5.21: Definition of the `Event_t` struct type in *SmartOS*.

```
typedef struct {
    bool      Value;
    PriorityListID_t ListHead;
} Event_t;
```

Listing 5.22: Definition of the `Event_t` struct type in UPPAAL.

The `Task_t` struct needed the most work to translate, as can be seen in the differences between Listings 5.23 and 5.24. The `Context`, `Name`, `StackHighestAddress`, `StackLowestAddress` and `Entry` fields were removed, as they are not relevant in the model. As mentioned before, the `TimeoutList` and `TimeoutListHead` were removed, as the time-outs are abstracted away in the models. This only leaves some pointer to ID translations, which can be easily done.


```

typedef struct Task_s {
    Word_t*           Context;
    char*             Name;
    Word_t*           StackHighestAddress;
    Word_t*           StackLowestAddress;
    Priority_t         BasePriority;
    Priority_t         CurrentPriority;
    Time_t            Timeout;
    void              (*Entry)(void);
    ListElement_t     PriorityList;
    ListElement_t     TimeoutList;
    struct Task_s**   PriorityListHead;
    struct Task_s**   TimeoutListHead;
    TaskResourceManagementStruct_t ResourceManagement;
} Task_t;

```

Listing 5.23: Definition of the `Task_t` struct type in *SmartOS*.

```

typedef struct {
    Priority_t         BasePriority;
    Priority_t         CurrentPriority;
    ListElement_t     PriorityList;
    PriorityListID_t   PriorityListHeadID;
    TaskResourceManagementStruct_t ResourceManagement;
} Task_t;

```

Listing 5.24: Definition of the `Task_t` struct type in *UPPAAL*.

The structs `Resource_t` and `TaskResourceManagementStruct_t` are part of the common interface described in Section 5.1.3 and, therefore, differ depending on the protocol.

HLP

The translation for the HLP structures consists only of some pointer to ID translations, as can be seen in Listings 5.25, 5.26, 5.27 and 5.28. The **Counter** field in the **Resource_t** struct in UPPAAL is bounded by the **ResourceCounterMax** constant as described in Section 5.2.2, instead of having the full range of **uint8_t**.

```
typedef struct Resource_s{
    uint8_t      Counter;
    struct Task_s* Owner;
    Priority_t    PrioCeiling;
} Resource_t;
```

Listing 5.25: Definition of the **Resource_t** struct type in *SmartOS* for HLP.

```
typedef struct {
    int [0, ResourceCounterMax] Counter;
    TaskID_t                      Owner;
    Priority_t                     PrioCeiling;
} Resource_t;
```

Listing 5.26: Definition of the **Resource_t** struct type in UPPAAL for HLP.

```
typedef struct {
    uint8_t OwnedResources;
} TaskResourceManagementStruct_t;
```

Listing 5.27: Definition of the **TaskResourceManagementStruct_t** struct type in *SmartOS* for HLP.

```
typedef struct {
    int [0, NumberResources] OwnedResources;
} TaskResourceManagementStruct_t;
```

Listing 5.28: Definition of the `TaskResourceManagementStruct_t` struct type in UPPAAL for HLP.

PCP

The translation for the `Resoure_t` and `TaskResourceManagementStruct_t` for PCP only consists of pointer to ID translation and the same bounding of the `Counter` field, as for HLP. The structures can be seen in Listings 5.29, 5.30, 5.31 and 5.32.

```
typedef struct Resource_s
{
    uint8_t      Counter;
    struct Task_s* Owner;
    Priority_t    PrioCeiling;
} Resource_t;
```

Listing 5.29: Definition of the `Resource_t` struct type in *SmartOS* for PCP.

```
typedef struct {
    int [0, ResourceCounterMax] Counter;
    TaskID_t                      Owner;
    Priority_t                     PrioCeiling;
} Resource_t;
```

Listing 5.30: Definition of the `Resource_t` struct type in UPPAAL for PCP.

```
typedef struct
{
    struct Resource_s* BlockingResource;
    Priority_t         TaskPrioCeiling;
} TaskResourceManagementStruct_t;
```

Listing 5.31: Definition of the `TaskResourceManagementStruct_t` struct type in *SmartOS* for PCP.

```
typedef struct {
    ResourceID_t BlockingResource;
    Priority_t    TaskPrioCeiling;
} TaskResourceManagementStruct_t;
```

Listing 5.32: Definition of the `TaskResourceManagementStruct_t` struct type in UPPAAL for PCP.

PIP

The translation for PIP works the same as for the other two protocols. The structures can be seen in Listings 5.33, 5.34, 5.35 and 5.36.

```
typedef struct Resource_s {
    uint8_t      Counter;
    struct Task_s* Owner;
    Priority_t    Priority;
    Event_t      Event;
} Resource_t;
```

Listing 5.33: Definition of the `Resource_t` struct type in *SmartOS* for PIP.

```
typedef struct {
    int [0, ResourceCounterMax] Counter;
    TaskID_t Owner;
    Priority_t Priority;
    Event_t Event;
} Resource_t;
```

Listing 5.34: Definition of the **Resource_t** struct type in UPPAAL for PIP.

```
typedef struct {
    struct Resource_s* BlockingResource;
} TaskResourceManagementStruct_t;
```

Listing 5.35: Definition of the **TaskResourceManagementStruct_t** struct type in *SmartOS* for PIP.

```
typedef struct {
    ResourceID_t BlockingResource;
} TaskResourceManagementStruct_t;
```

Listing 5.36: Definition of the **TaskResourceManagementStruct_t** struct type in UPPAAL for PIP.

5.3.4 Functions

The translation of functions follows some basic rules, described with an example in the following sections. Nearly everything could be translated using those simple rules. The most notable exceptions were some complicated loops, which sometimes needed extensive rewriting, though they were uncommon. Some code segments that use some form of optimization using data structures that are abstracted away also needed some rewriting, as the data structures no longer existed.

Pointers

As described in Section 5.3.1, all pointers needed to be translated to corresponding IDs. This needed to be done for the parameters and all pointers used within the function. An example of the translation of parameters can be found in Listings 5.37 and 5.38. This example also shows the translation of pointers to pointers to IDs indexing an array of IDs. These translations were handled using an array of IDs. As the relevant parts of *SmartOS* only ever use pointer-pointers for saving the references to the head of a priority list, the **PriorityListHeads** array with its associated IDs shown in Listings 5.15 and 5.16 was enough to accomplish these translations. This translation gets very tricky as there are multiple ID types per pointer type because of the differentiation between IDs that are allowed to be the "illegal" value or not.

```
void os_insertPriority(Task_t* Task, Task_t** ListHead);
```

Listing 5.37: Example of a translation of pointers in parameters of functions in C.

```
void os_insertPriority(TaskID_t TaskID, PriorityListID_t  
    PriorityListHeadID);
```

Listing 5.38: Example of a translation of pointers in parameters of functions in UPPAAL.

Another often used feature of pointers is the check if they are valid, i.e., not **NULL**. This can be translated using the "illegal" value for each ID type, as described in Section 5.3.1. An example of this can be seen in Listings 5.39 and 5.40.

```
while(Curr != NULL)  
    ...
```

Listing 5.39: Example of a translation of checking validity of pointers in C.

```
while (CurrID != IllegalTaskID)
    ...
```

Listing 5.40: Example of a translation of checking validity of pointers in UPPAAL.

Pointers often refer to structs. As referencing fields using the `->` operator in C no longer works, these references need to be translated to array accesses using IDs as indices. This can be seen in Listings 5.41 and 5.42.

```
Curr = Curr->PriorityList.Next;
```

Listing 5.41: Example of a translation of referencing fields in pointers to structs in C.

```
CurrID = Tasks[CurrID].PriorityList.Next;
```

Listing 5.42: Example of a translation of referencing fields in pointers to structs in UPPAAL.

Whenever a variable is dereferenced, it needs to be transformed to an array access with an ID, similar to the struct field access above. an example of this can be seen in Listings 5.43 and 5.44.

```
*ListHead = Task;
```

Listing 5.43: Example of a translation of dereferencing pointers in C.

```
PriorityListHeads[PriorityListHeadID] = TaskID;
```

Listing 5.44: Example of a translation of dereferencing pointers in UPPAAL.

Whenever a variable is referenced, i.e., transformed to a pointer, it needs to be changed to an ID. An example of this can be seen in Listings 5.45 and 5.46.

```
if (ListHead == &os_ReadyQueue)
    ...
```

Listing 5.45: Example of a translation of dereferencing pointers in C.

```
if (PriorityListHeadID == os_ReadyQueueID)
    ...
```

Listing 5.46: Example of a translation of dereferencing pointers in UPPAAL.

Local variables that are pointers need to be changed to their corresponding IDs. An example of this can be seen in Listings 5.47 and 5.48.

```
Task_t* Curr;
```

Listing 5.47: Example of a translation of pointers as local variables in C.

```
TaskID_t CurrID;
```

Listing 5.48: Example of a translation of pointers as local variables in UPPAAL.

Local Variable Declaration

Local variables in UPPAAL functions need to be declared at the beginning of the function. As this is not the case in modern C, the declaration of the variables has to be moved. Instances where the variables are declared and directly set to a specific value are split into a declaration in the beginning and the assignment at the original location in the code. An example of this can be seen in Listings 5.49 and 5.50.


```
Task_t* Curr = *ListHead;
```

Listing 5.49: Example of splitting a local variable declaration in C.

```
TaskID_t Curr;
...
Curr = PriorityListHeads[PriorityListHeadID];
```

Listing 5.50: Example of splitting a local variable declaration in UPPAAL.

Timing

As mentioned before and explained in Section 5.2.1, everything related to timing was removed from the model. This step can not be easily automated, but it was a rule followed during the manual translation. Examples range from removing parameters from functions, as seen in Listings 5.51 and 5.52, to removing whole code sections.

```
void syscall_getResourceUntil(Resource_t* Resource, Time_t* Deadline);
```

Listing 5.51: Example of removing time related function parameters in C.

```
void syscall_getResourceUntil(ResourceIDLegal_t ResourceID);
```

Listing 5.52: Example of removing time related function parameters in UPPAAL.

Return Values of Syscalls

As syscalls in *SmartOS* need to save their return values in the context of the Task, which does not exist in the model, and the return values are irrelevant, the statements can be removed. An example of such a statement can be seen in Listing 5.53.

```
MyTask->Context[CTX_RETVAL0] = SUCCESS;
```

Listing 5.53: Example of saving the return value of a syscall in the context of a Task in C.

Assertions

SmartOS uses different kinds of assertions to ensure the Tasks are using the syscalls correctly and that all kernel data structures are consistent. These statements can be removed when transforming to UPPAAL code. Some assertions can be used as inspiration for guards in the templates or for queries in the verifier. Listing 5.54 shows an example of such a statement.

```
OS_ASSERT(Task->BasePriority != 0, "Task == IdleTask");
```

Listing 5.54: Example of an assertion in C that was used as inspiration for a guard.

`min()` and `max()`

The C function `min()` and `max` do not exist in UPPAAL. Instead, there exist the special operators, `<?` and `>?`, that have the same functionality. An example of the translation can be seen in Listings 5.55 and 5.56.

```
NewP = max(NewP, os_Resources[Index].PrioCeiling);
```

Listing 5.55: Example of the use of `max` in C.

```
NewP = NewP >? Resources[Index].PrioCeiling;
```

Listing 5.56: Example of the translation of `max` using the `>?` operator in UPPAAL.

Special Operators

Some C operators do not exist in UPPAAL. An example is the often used `++` operator. Statements using these operators have to be written out. An example can be seen in Listings 5.57 and 5.58.

```
Index++;
```

Listing 5.57: Example of the use of a special operator in C.

```
Index = Index + 1;
```

Listing 5.58: Example of the translation of a special operator in UPPAAL.

Implicit `if`-Statements

In C, using an `if` clause with an integer does not need the `!= 0` statement. This is not the case for UPPAAL, where this always needs to be written. An example of such a statement can be seen in Listings 5.59 and 5.60.

```
if(Task->ResourceManagement.OwnedResources)
    ...
```

Listing 5.59: Example of the use of an implicit `if` statement in C.

```
if (Tasks[TaskID].ResourceManagement.OwnedResources > 0)
    ...
```

Listing 5.60: Example of the translation implicit `if` statement in UPPAAL.

Loops with Iterators

Some loops iterate over all Tasks or Resources using pointers. These loops could be easily rewritten using for loops over the range of the IDs of the respective data type. An example of this can be seen in Listings 5.61 and 5.62.

```
Task_t* TaskIterator = (Task_t*)os_TasksBegin;
while (TaskIterator < (Task_t*)os_TasksEnd)
{
    ...
    TaskIterator++;
}
```

Listing 5.61: Example of a loop over all Tasks in C.

```
for (TaskIteratorID : TaskIDProper_t)
{
    ...
}
```

Listing 5.62: Example of a loop over all Tasks in UPPAAL.

Helper Variables

PCP and PIP need two helper variables for the model to be set in the code. The two helper variables can be seen in Listing 5.63.

```
bool helper_ResourceEventSet;
TaskID_t helper_ResourceEventSetTaskID;
```

Listing 5.63: Helper variables for PCP and PIP needed in UPPAAL.

CHAPTER 6

Conclusion and Future Work

This chapter first answers the research question formulated in Chapter 1. Afterwards, it lists some other achievements of this thesis's work. It concludes with some ideas for future work using the results of the thesis.

6.1 Answering the Research Question

An adequate answer to the question

Is it possible to create a UPPAAL model from C code in a way that can be automated?

can not be formulated with a clear and simple answer, as the answer highly depends on what creating an UPPAAL model entails.

Suppose it is only translating functions written in C to UPPAAL and calling those functions via updates in an existing automaton. In that case, yes, it can be automated using the rules outlined in Section 5.3. Even though the rules were formulated for the very specific coding style of the author, one can argue that this would be possible for every style of C code. Writing the C code with the translation to UPPAAL in mind and knowing all the limitations of the UPPAAL C-like language would simplify the formulation further. The translation of the last protocol, the PCP, only took about two hours, as the templates of the PIP could be reused, and all the rules were already written down and easy to follow. Getting the model to work and finding the few bugs the protocol still had were different stories, taking significantly more effort.

But having the templates of the needed automata already in place is a luxury that usually is not the case. Creating those templates is the hardest part of creating a representation of the

protocols in UPPAAL. This is nearly impossible to do in an automated way, as it requires many decisions on what to include in the model and what parts to abstract or leave out entirely. For these decisions, a deep understanding of the underlying system is needed, which goes beyond understanding the C code. One could argue that a sufficiently big model of *SmartOS* that would depict all possible control flows could be used. This model might be more straightforward to create automatically but would not be usable as the state space would explode, and no verification could ever be done in a reasonable time. There is a reason why the basic model of *SmartOS* used in this thesis is as small as it is. And even with this small model, simulating more than 4 Tasks and Resources was impossible for PCP on a Laptop with 8 GB of RAM. But as memory usage grows exponentially with the number of Tasks, even powerful machines would not have managed much more. This need to minimize the model is also why the UPPAAL models of Batista Ribeiro et al. [24] could not be reused, and a new base model had to be created from scratch. So if the templates are not given, *no*, it is not feasible to create a meaningful model from the C code in an automated way.

6.2 Secondary Goals

The work done in the scope of this thesis produced more contributions than just answering the research question.

First, *SmartOS* now features three different resource management systems that can be easily switched between without changing anything in the application code.

Secondly, the base UPPAAL model created for this thesis can be used as a starting point for developing models that depict different aspects of *SmartOS*, as it is small enough to be extended without becoming too large to be practical.

The created models of the different protocols might not be qualified to prove the correctness of the C code formally. Still, they were sufficient to find bugs not revealed during implementation testing. This shows that even if a formal proof of code is not needed, models can still play an essential role in enhancing code quality. Finding and fixing these bugs in *SmartOS* is this thesis's third and most crucial secondary achievement.

6.3 Future Work

There exist lots of points where future work could build upon this thesis.

Even though a complete automatic transformation from code to model is not feasible,

translating C code to code usable by UPPAAL is possible using the rules outlined in the thesis. Complementing the rules with adequate coding guidelines for *SmartOS* could further increase the potency of the translations and open up the translation of even more parts of the RTOS.

The base model of *SmartOS* developed for this thesis can be used as a baseline to model additional parts of the OS kernel. This could lead to a complete model of the kernel split into smaller models that are still possible to verify in a reasonable time. The smaller models would feature a detailed model of a kernel section, with the other parts abstracted away. This can be similar to how Events are abstracted away in this thesis while detailing the resource management part of the kernel.

Another possible direction for further research can be the addition of clocks to the models to track the execution times of Tasks and do schedulability analysis using the different protocols. This becomes especially interesting when adding timing characterizations of the protocols, allowing the models to become powerful tools in selecting the most suited protocol for a specific Task set.

Creating the templates for the automata has proven difficult to do in a way that could be easily automated. This does not mean it is impossible to derive a UPPAAL model from the *SmartOS* source code. Adding specific markers in the source code might help to create a better one-to-one correlation between the execution flow of the source code and the templates by allowing the automaton locations to correspond to a specific line in the code.

Furthermore, the models of Batista Ribeiro et al. [24] and the models presented in this thesis could be merged. This would lead to a large model and long verification times, but it might be doable on large enough machines. If adding the more complex resource management protocols proves not feasible, the lessons learned from this thesis still might enable the usage of directly translated source code in those models, improving their ability to reflect *SmartOS*.

Bibliography

- [1] OSEK-VDX, “OSEK-VDX.” <https://www.osek-vdx.org/>, 2024. [Online; accessed 06-January-2024]. → [p1], [p11]
- [2] AUTOSAR, “AUTOSAR.” <https://www.autosar.org/>, 2024. [Online; accessed 06-January-2024]. → [p1], [p11]
- [3] AUTOSAR, “AUTOSAR Acceptance Tests.” <https://www.autosar.org/standards/archive/acceptance-tests-for-classic-platform>, 2024. [Online; accessed 06-January-2024]. → [p1], [p12]
- [4] The RTEMS Project, “RTEMS.” <https://www.rtems.org/>, 2024. [Online; accessed 06-January-2024]. → [p1], [p11]
- [5] Frontgrade Gaisler, “GR712RC.” <https://www.gaisler.com/index.php/products/components/gr712rc>, 2023. [Online; accessed 08-January-2024]. → [p1]
- [6] Frontgrade Gaisler, “GR740.” <https://www.gaisler.com/index.php/products/components/gr740>, 2023. [Online; accessed 08-January-2024]. → [p1]
- [7] R. M. Gomes and M. Baunach, “A framework for os portability: From formal models to low-level code,” in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, SAC ’22, (New York, NY, USA), p. 1156–1165, Association for Computing Machinery, 2022. → [p2], [p12]
- [8] T. Scheipel, L. Batista Ribeiro, T. Sagaster, and M. Baunach, “SmartOS: An OS Architecture for Sustainable Embedded Systems.” Tagungsband des FG-BS Frühjahrstreffens 2022, 2022. → [p5], [p12], [p26]
- [9] UPPAAL, “UPPAAL.” <https://uppaal.org/>, 2024. [Online; accessed 06-January-2024]. → [p7]

- [10] UPPAAL, “UPPAAL Documentation.” <https://docs.uppaal.org/>, 2024. [Online; accessed 06-January-2024]. → [p7]
- [11] OSEK-VDX, “Specification OSEK OS 2.2.3,” 2005. → [p11]
- [12] AUTOSAR, “Specification of Operating System AUTOSAR CP R23-11.” https://www.autosar.org/fileadmin/standards/R23-11/CP/AUTOSAR_CP_SWS_OS.pdf, 2023. [Online; accessed 06-January-2024]. → [p11]
- [13] European Space Agency, “RTEMS SMP Qualification Data Pack.” <https://rtems-qual.io.esa.int/>, 2022. [Online; accessed 06-January-2024]. → [p11]
- [14] The RTEMS Project, “RTEMS Classic API Guide 6.e4a0de6.” <https://docs.rtems.org/branches/master/c-user/index.html>, 2023. [Online; accessed 06-January-2024]. → [p11]
- [15] FreeRTOS, “FreeRTOS - Market Leading RTOS.” <https://www.freertos.org/index.html>, 2024. [Online; accessed 06-January-2024]. → [p11]
- [16] STMicroelectronics, “STMicroelectronics.” https://www.st.com/content/st_com/en.html, 2023. [Online; accessed 06-January-2024]. → [p11]
- [17] STMicroelectronics, “X-CUBE-FREERTOS.” <https://www.st.com/en/embedded-software/x-cube-freertos.html>, 2023. [Online; accessed 06-January-2024]. → [p12]
- [18] FreeRTOS, “FreeRTOS Mutexes.” <https://www.freertos.org/Real-time-embedded-RTOS-mutexes.html>, 2024. [Online; accessed 06-January-2024]. → [p12]
- [19] Zephyr Project, “The Zephyr Project - A proven RTOS ecosystem, by developers, for developers.” <https://www.zephyrproject.org/>, 2024. [Online; accessed 06-January-2024]. → [p12]
- [20] Zephyr Project, “Mutexes - Zephyr Project Documentation.” <https://docs.zephyrproject.org/latest/kernel/services/synchronization/mutexes.html>, 2024. [Online; accessed 06-January-2024]. → [p12]
- [21] L. Batista Ribeiro, F. Schlager, and M. Baunach, “Towards automatic sw integration in dependable embedded systems,” in *EWSN - International Conference on Embedded Wireless Systems and Networks*, 02 2020. → [p12]

- [22] T. Scheipel, F. Angermair, and M. Baunach, “moremcu: A runtime-reconfigurable risc-v platform for sustainable embedded systems,” in *2022 25th Euromicro Conference on Digital System Design (DSD)*, 08 2022. → [p12]
- [23] M. Malenko and M. Baunach, *Hardware/Software Co-designed Security Extensions for Embedded Devices*, pp. 3–14. Springer International Publishing, 04 2019. → [p12]
- [24] L. Batista Ribeiro, F. Lorber, U. Nyman, K. Larsen, and M. Baunach, *A Modeling Concept for Formal Verification of OS-Based Compositional Software*, pp. 26–46. Springer Nature Switzerland, 04 2023. → [p12], [p28], [p60], [p61]
- [25] M. Lindahl, P. Pettersson, and W. Yi, “Formal Design and Analysis of a Gearbox Controller,” *Springer International Journal of Software Tools for Technology Transfer (STTT)*, vol. 3, no. 3, pp. 353–368, 2001. → [p12]
- [26] L. Khatib, N. Muscettola, and K. Havelund, “Verification of plan models using uppaal,” 02 2001. → [p12]
- [27] L. Shan, Y. Wang, N. Fu, X. Zhou, L. Zhao, L. Wan, L. Qiao, and J. Chen, “Formal verification of lunar rover control software using uppaal,” in *FM 2014: Formal Methods* (C. Jones, P. Pihlajasaari, and J. Sun, eds.), (Cham), pp. 718–732, Springer International Publishing, 2014. → [p12]
- [28] A. Hendrawan and D. Adzkiya, “Modelling and verification of cash withdrawal transaction in automated teller machine using timed automata,” *Journal of Physics: Conference Series*, vol. 1821, p. 012031, 03 2021. → [p12]
- [29] A. David, J. Rasmussen, K. Larsen, and A. Skou, “Model-based framework for schedulability analysis using uppaal 4.1,” 11 2009. → [p12]
- [30] L. Shan and S. Graf, “Timing verification of an aerial video tracking system using uppaal,” 2015. → [p12]
- [31] M. Mikučionis, K. G. Larsen, J. I. Rasmussen, B. Nielsen, A. Skou, S. U. Palm, J. S. Pedersen, and P. Hougard, “Schedulability analysis using uppaal: Herschel-planck case study,” in *Leveraging Applications of Formal Methods, Verification, and Validation* (T. Margaria and B. Steffen, eds.), (Berlin, Heidelberg), pp. 175–190, Springer Berlin Heidelberg, 2010. → [p12]

- [32] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, “Sel4: Formal verification of an os kernel,” in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP '09*, (New York, NY, USA), p. 207–220, Association for Computing Machinery, 2009. → [p12]
- [33] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, “Comprehensive formal verification of an os microkernel,” vol. 32, feb 2014. → [p12]
- [34] D. B. de Oliveira, T. Cucinotta, and R. S. de Oliveira, “Efficient formal verification for the linux kernel,” in *Software Engineering and Formal Methods* (P. C. Ölveczky and G. Salaün, eds.), (Cham), pp. 315–332, Springer International Publishing, 2019. → [p12]
- [35] S. Bhat and H. Shacham, “Formal verification of the linux kernel ebpf verifier range analysis,” 2022. → [p12]
- [36] S.-W. Li, X. Li, R. Gu, J. Nieh, and J. Zhuang Hui, “A secure and formally verified linux kvm hypervisor,” in *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 1782–1799, 2021. → [p12]
- [37] G. Holzmann and S. Margaret, “Software model checking: Extracting verification models from source code,” *Softw. Test., Verif. Reliab.*, vol. 11, pp. 65–79, 06 2001. → [p12]
- [38] European Cooperation for Space Standardization, “ECSS-E-ST-40C – Software,” 2009. → [p12]
- [39] European Cooperation for Space Standardization, “ECSS-Q-ST-80C Rev.1 – Software product assurance,” 2017. → [p12]
- [40] Simulink Documentation, “Simulation and model-based design,” 2024. → [p12]
- [41] J. Krizan, L. Ertl, M. Bradac, M. Jasansky, and A. Andreev, “Automatic code generation from matlab/simulink for critical applications,” in *2014 IEEE 27th Canadian Conference on Electrical and Computer Engineering (CCECE)*, pp. 1–6, 2014. → [p12]
- [42] B. W. Kernighan and D. M. Ritchie, *The C programming language*. Prentice Hall, 2006. → [p21]
- [43] GNU Compiler Collection Team, “GCC, the GNU Compiler Collection.” <https://gcc.gnu.org/>, 2023. [Online; accessed 08-January-2024]. → [p23]

- [44] LLVM Foundation, “Clang C Language Family Frontend for LLVM.” <https://clang.llvm.org/>, 2023. [Online; accessed 08-January-2024]. → [p23]
- [45] L. B. Ribeiro, D. Nagarajan, V. Manjunath, M. T. Ali Ahmad, and M. Baunach, “Verifying liveness and real-time of os-based embedded software,” in *2022 25th Euromicro Conference on Digital System Design (DSD)*, pp. 679–688, 2022. → [p28]

List of Figures

2.1	The different types of locations in UPPAAL.	8
2.2	The different annotations of an edge in UPPAAL.	9
4.1	Parallel timeline of the example, where all Tasks get executed simultaneously.	19
4.2	The timeline using the HLP, showing the priority changes during the execution of the three Tasks.	20
4.3	The timeline using the PCP, showing the priority changes during the execution of the three Tasks.	20
4.4	The timeline using the PIP, showing a deadlock scenario where both Task L, in green, and Task H, in red, are blocked forever.	20
5.1	The directory tree of the relevant directories and files for the resource management protocols.	22
5.2	Template for the Task automata of the base model. The template has one parameter: <code>const TaskIDLegal_t TaskID</code>	27
5.3	Template for the Kernel automata of the base model. The template has no parameters.	30
5.4	Template for the Task automata of the HLP model. The template has one parameter: <code>const TaskIDLegal_t TaskID</code>	31
5.5	Template for the Kernel automaton of the HLP model. The template has no parameters.	34
5.6	The first part of the template for the Task automata of the PCP model. The template has one parameter: <code>const TaskIDLegal_t TaskID</code> . The model misses edges between the Ready and Running Location as well as the Waiting location and all its edges shown in Figure 5.7.	36
5.7	The second part of the template for the Task automata of the PCP model. This shows the remaining edges and locations of Figure 5.6	37

5.8	Template for the Kernel automaton of the PCP model. The template has no parameters.	38
5.9	The first part of the template for the Task automata of the PIP model. The template has one parameter: const TaskIDLegal_t TaskID . The model misses edges between the Ready and Running Location as well as the Waiting location and all its edges shown in Figure 5.10.	40
5.10	The second part of the template for the Task automata of the PIP model. This shows the remaining edges and locations of Figure 5.9	41
5.11	Template for the Kernel automaton of the PIP model. The template has no parameters.	42

List of Tables

4.1	The timeline of Task L.	18
4.2	The timeline of Task M.	18
4.3	The timeline of Task H.	19

List of Listings

2.1	Definition of the integer type Type_t with the inclusive lower bound lowerBound and the inclusive upper bound upperBound . The two variables used for the bounds must be of the type const int	9
5.1	Code used to include the os_resource_types.h file.	23
5.2	Tries to get a Resource. If the Resource is unavailable, the Task waits until the absolute deadline Deadline is reached. For HLP the deadline is ignored, as the Task will always get the Resource without waiting	24
5.3	Same functionality as getResourceUntil but the Task will wait indefinitely for the Resource if it is not freed.	24
5.4	The Task releases the Resource.	24
5.5	The syscall for getting a Resource with a deadline (deadline ignored for HLP).	24
5.6	The syscall for releasing a Resource.	24
5.7	A check done when a Task tries to go to sleep. Important for HLP as Tasks are not allowed to sleep while holding a Resource. For PCP and PIP this function always returns SUCCESS as the protocols allow for Tasks to sleep at any time.	24
5.8	A handler called when a Task, which was waiting for an Event, timed out. The Event might have belonged to the resource management protocol and needs to be handled differently depending on the protocol used. For HLP, it simply removes the Task from the queue waiting for the Event, as HLP does not need any Events.	25
5.9	Initializes the protocol-specific data structure for Tasks. This has to be done during the startup procedure, as the TCBs only gets created at that time.	25
5.10	This function gets called during startup. HLP and PCP calculate the needed ceiling priorities during that call. PIP does not need to do anything here, as no ceiling priorities are needed.	25
5.11	The central data structure that needs to contain all the data of a Resource, needed for the syscalls.	25

5.12	Data structure that is part of the TCB and contains all the Task-specific data needed for the protocol.	25
5.13	Macro to create a new Resource. Allocates and initializes the needed data structures.	26
5.14	Macro to register a Task Resource pair.	26
5.15	Different ID types used to replace pointers. Not all protocols use all ID types, e.g., HLP does not use ResourceID_t	44
5.16	Arrays used in combination with IDs to replace pointers.	44
5.17	Definition of the Priority_t type in <i>SmartOS</i>	45
5.18	Definition of the Priority_t type in the UPPAAL models.	45
5.19	Definition of the ListElement_t struct type in <i>SmartOS</i>	45
5.20	Definition of the ListElement_t struct type in UPPAAL.	45
5.21	Definition of the Event_t struct type in <i>SmartOS</i>	46
5.22	Definition of the Event_t struct type in UPPAAL.	46
5.23	Definition of the Task_t struct type in <i>SmartOS</i>	47
5.24	Definition of the Task_t struct type in UPPAAL.	47
5.25	Definition of the Resource_t struct type in <i>SmartOS</i> for HLP.	48
5.26	Definition of the Resource_t struct type in UPPAAL for HLP.	48
5.27	Definition of the TaskResourceManagementStruct_t struct type in <i>SmartOS</i> for HLP.	48
5.28	Definition of the TaskResourceManagementStruct_t struct type in UPPAAL for HLP.	49
5.29	Definition of the Resource_t struct type in <i>SmartOS</i> for PCP.	49
5.30	Definition of the Resource_t struct type in UPPAAL for PCP.	49
5.31	Definition of the TaskResourceManagementStruct_t struct type in <i>SmartOS</i> for PCP.	50
5.32	Definition of the TaskResourceManagementStruct_t struct type in UPPAAL for PCP.	50
5.33	Definition of the Resource_t struct type in <i>SmartOS</i> for PIP.	50
5.34	Definition of the Resource_t struct type in UPPAAL for PIP.	51
5.35	Definition of the TaskResourceManagementStruct_t struct type in <i>SmartOS</i> for PIP.	51
5.36	Definition of the TaskResourceManagementStruct_t struct type in UPPAAL for PIP.	51
5.37	Example of a translation of pointers in parameters of functions in C.	52

5.38	Example of a translation of pointers in parameters of functions in UPPAAL. . .	52
5.39	Example of a translation of checking validity of pointers in C.	52
5.40	Example of a translation of checking validity of pointers in UPPAAL.	53
5.41	Example of a translation of referencing fields in pointers to structs in C.	53
5.42	Example of a translation of referencing fields in pointers to structs in UPPAAL.	53
5.43	Example of a translation of dereferencing pointers in C.	53
5.44	Example of a translation of dereferencing pointers in UPPAAL.	53
5.45	Example of a translation of dereferencing pointers in C.	54
5.46	Example of a translation of dereferencing pointers in UPPAAL.	54
5.47	Example of a translation of pointers as local variables in C.	54
5.48	Example of a translation of pointers as local variables in UPPAAL.	54
5.49	Example of splitting a local variable declaration in C.	55
5.50	Example of splitting a local variable declaration in UPPAAL.	55
5.51	Example of removing time related function parameters in C.	55
5.52	Example of removing time related function parameters in UPPAAL.	55
5.53	Example of saving the return value of a syscall in the context of a Task in C. .	56
5.54	Example of an assertion in C that was used as inspiration for a guard.	56
5.55	Example of the use of <code>max</code> in C.	56
5.56	Example of the translation of <code>max</code> using the <code>>?</code> operator in UPPAAL.	56
5.57	Example of the use of a special operator in C.	57
5.58	Example of the translation of a special operator in UPPAAL.	57
5.59	Example of the use of an implicit <code>if</code> statement in C.	57
5.60	Example of the translation implicit <code>if</code> statement in UPPAAL.	57
5.61	Example of a loop over all Tasks in C.	58
5.62	Example of a loop over all Tasks in UPPAAL.	58
5.63	Helper variables for PCP and PIP needed in UPPAAL.	58

List of Abbreviations

AUTOSAR "AUTomotive Open System ARchitecture"

EAS Embedded Automotive Systems

ECSS European Cooperation for Space Standardization

ESA European Space Agency

HLP Highest Locker Protocol

ICPP Immediate Ceiling Priority Protocol

IDE Integrated Development Environment

MrsP Multiprocessor Resource Sharing Protocol

OMIP O(m) Independence-Preserving Protocol

OS Operating System

OSEK "Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug"

PCP Priority Ceiling Protocol

PIP Priority Inheritance Protocol

RTEMS Real-Time Executive for Multiprocessor Systems

RTOS Real-Time Operating System

TCB Task Control Block

I love deadlines. I love the whooshing noise they make as they go by. – Douglas Adams