PALS/PRISM
Software Design Description (SDD): Ver. 0.51

Prepared by Cheolgi Kim, Abdullah Al-Nayeem, Heechul Yun,
Po-Liang Wu, and Lui Sha

of CS Dept., University of Illinois at Urbana-Champaign,

in Collaboration with
S. Bray, A. Carnifax, R. S. Hickey, K. D. Laviolette,
J. A. Lock, T. M. Parks and M. E. Ucal

of Lockheed Martin Co.

August 11, 2011

Contents

I Software Design Description **13**

1 Overview **14**
1.1 Introduction 14
1.1.1 Purpose 17
1.1.2 Scope 17
1.2 System Overview 17

2 PRISM Design and Definition **20**
2.1 System Overview 20
2.1.1 Lockstep synchronization 20
2.1.2 Harmonic synchronization 22
2.1.3 Client-server synchronization 23
2.1.4 Intra-period synchronization 25
2.1.5 Environment input/output synchronizer 25
2.2 Hardware Architecture 26
2.3 Support Software 27
2.4 Implementation Details 27
2.5 Software Scalability and Flexibility 27

3 System Architectural Design **29**
3.1 System Components 31
3.1.1 `Prism_manager` and `Abstract_thread_factory` 31
3.1.2 `Prism_task` 31
3.1.3 `TX_prism_port` 32
3.1.4 `RX_prism_port` 32
3.2 Concept of Execution 33
3.3 Interface Design 34
3.4 Software Development Environment 34
3.4.1 Compilers 34

3.4.2 Operating Systems 34

4 PRISM Detailed Design **35**
4.1 Class Diagram for PRISM core 35
4.2 Class Diagram for system dependent part 36
4.2.1 System-dependent communications 36
4.2.2 System-dependent thread interfaces 37
4.3 Activities between `Prism_task` class and its subclass objects . 39
4.4 Interactions between `Prism_task` and `TX_prism_port` 40
4.5 Class and Use Cases for `Prism_task` and `RX_prism_port` . . . 42
4.6 Pure PALS and Client-server semantics 44

II Reference Manual **47**

5 Class Index **48**
5.1 Class Hierarchy 48

6 Class Index **50**
6.1 Class List 50

7 Class Documentation **52**
7.1 prism::Abstract_condition_variable Class Reference 52
7.1.1 Detailed Description 53
7.1.2 Member Function Documentation 53
7.1.2.1 signal 53
7.1.2.2 wait 53
7.2 prism::Abstract_lock Class Reference 54
7.2.1 Detailed Description 54
7.2.2 Member Function Documentation 55
7.2.2.1 lock 55
7.2.2.2 trylock 55
7.2.2.3 unlock 55
7.3 prism::Abstract_thread_engine Class Reference 55
7.3.1 Detailed Description 56
7.3.2 Member Function Documentation 57
7.3.2.1 start 57
7.3.2.2 wait_for_completion 57

7.4 prism::Abstract_thread_factory Class Reference 58
7.4.1 Detailed Description 59
7.4.2 Member Function Documentation 59
7.4.2.1 create_condition_variable 59
7.4.2.2 create_lock 59
7.4.2.3 create_thread_engine 60
7.4.2.4 create_timer 60
7.4.2.5 get_current_time 60
7.5 prism::Abstract_timer Class Reference 61
7.5.1 Detailed Description 61
7.5.2 Member Function Documentation 62
7.5.2.1 restart_timer 62
7.5.2.2 stop_timer 62
7.5.2.3 wait 62
7.6 prism::Lock_failed Class Reference 63
7.6.1 Detailed Description 63
7.7 prism::Port_buffer Class Reference 63
7.7.1 Detailed Description 66
7.7.2 Constructor & Destructor Documentation 66
7.7.2.1 Port_buffer 66
7.7.3 Member Function Documentation 67
7.7.3.1 append_double 67
7.7.3.2 append_float 67
7.7.3.3 get_max_size 67
7.7.3.4 get_remained_size 67
7.7.3.5 retrieve_double 68
7.7.3.6 retrieve_float 68
7.7.3.7 size 68
7.7.3.8 tostr 68
7.8 prism::Port_buffer_overflow Class Reference 69
7.8.1 Detailed Description 69
7.9 prism::Port_exception Class Reference 69
7.9.1 Detailed Description 70
7.10 prism::Port_no_data_received Class Reference 70
7.10.1 Detailed Description 70
7.11 prism::POSIX_condition_variable Class Reference 70
7.11.1 Detailed Description 72
7.12 prism::POSIX_lock Class Reference 72

7.12.1 Detailed Description 74
7.12.2 Constructor & Destructor Documentation 74
7.12.2.1 POSIX_lock 74
7.12.2.2 ∼POSIX_lock 74
7.12.3 Member Function Documentation 74
7.12.3.1 lock 74
7.12.3.2 trylock 74
7.12.3.3 unlock 74
7.13 prism::POSIX_thread_engine Class Reference 75
7.13.1 Detailed Description 76
7.13.2 Member Function Documentation 77
7.13.2.1 start 77
7.13.2.2 wait_for_completion 77
7.14 prism::POSIX_thread_factory Class Reference 77
7.14.1 Detailed Description 79
7.14.2 Member Function Documentation 79
7.14.2.1 create_condition_variable 79
7.14.2.2 create_lock 80
7.14.2.3 create_thread_engine 80
7.14.2.4 create_timer 80
7.14.2.5 get_current_time 81
7.14.2.6 get_factory 81
7.15 prism::POSIX_timer Class Reference 81
7.15.1 Detailed Description 83
7.15.2 Constructor & Destructor Documentation 83
7.15.2.1 POSIX_timer 83
7.15.3 Member Function Documentation 84
7.15.3.1 restart_timer 84
7.15.3.2 stop_timer 84
7.15.3.3 wait 84
7.16 Print_buffer Class Reference 84
7.16.1 Detailed Description 85
7.17 prism::Prism_manager Class Reference 85
7.17.1 Detailed Description 86
7.17.2 Member Function Documentation 86
7.17.2.1 get_current_time 86
7.17.2.2 get_prism_default_factory 86
7.17.2.3 initialize_prism 87

7.17.2.4 is_prism_initialized 87
7.18 prism::Prism_task Class Reference 87
7.18.1 Detailed Description 90
7.18.2 Constructor & Destructor Documentation 90
7.18.2.1 Prism_task 90
7.18.2.2 Prism_task 90
7.18.2.3 Prism_task 91
7.18.3 Member Function Documentation 91
7.18.3.1 each_pals_period 91
7.18.3.2 get_base_time 91
7.18.3.3 get_pals_period 91
7.18.3.4 get_pals_period_in_ns 92
7.18.3.5 get_timer_index 92
7.18.3.6 initialize 92
7.18.3.7 run 92
7.18.3.8 wait_for_timer 92
7.19 prism::Prism_time Class Reference 93
7.19.1 Detailed Description 94
7.19.2 Constructor & Destructor Documentation 94
7.19.2.1 Prism_time 94
7.19.3 Member Function Documentation 94
7.19.3.1 add_nanoseconds 94
7.19.3.2 add_time 94
7.19.3.3 compare 95
7.19.3.4 get_nanosecond 95
7.19.3.5 get_second 95
7.19.3.6 subtract_nanoseconds 95
7.20 Realtime_fprintf Class Reference 96
7.20.1 Detailed Description 97
7.20.2 Friends And Related Function Documentation 97
7.20.2.1 initialize_rt_fprintf 97
7.20.2.2 rt_fprintf 98
7.21 prism::RX_port Class Reference 98
7.21.1 Detailed Description 99
7.21.2 Constructor & Destructor Documentation 99
7.21.2.1 RX_port 99
7.21.3 Member Function Documentation 100
7.21.3.1 recv 100

7.21.3.2 recv_port_buffer . . . 100
7.22 prism::RX_POSIX_multicast_port Class Reference . . . 101
7.22.1 Detailed Description . . . 102
7.22.2 Member Function Documentation . . . 102
7.22.2.1 recv . . . 102
7.23 prism::RX_POSIX_unicast_port Class Reference . . . 103
7.23.1 Detailed Description . . . 105
7.23.2 Member Function Documentation . . . 105
7.23.2.1 recv . . . 105
7.24 prism::RX_prism_port Class Reference . . . 106
7.24.1 Detailed Description . . . 107
7.24.2 Constructor & Destructor Documentation . . . 107
7.24.2.1 RX_prism_port . . . 107
7.24.2.2 RX_prism_port . . . 108
7.24.3 Member Function Documentation . . . 109
7.24.3.1 recv . . . 109
7.24.3.2 recv_port_buffer . . . 110
7.25 prism::Thread_stub Class Reference . . . 110
7.25.1 Detailed Description . . . 112
7.25.2 Constructor & Destructor Documentation . . . 112
7.25.2.1 Thread_stub . . . 112
7.25.3 Member Function Documentation . . . 113
7.25.3.1 run . . . 113
7.25.3.2 wait_for_completion . . . 113
7.26 prism::TX_port Class Reference . . . 113
7.26.1 Detailed Description . . . 114
7.26.2 Member Function Documentation . . . 114
7.26.2.1 send . . . 114
7.26.2.2 send_port_buffer . . . 115
7.27 prism::TX_POSIX_multicast_port Class Reference . . . 116
7.27.1 Detailed Description . . . 117
7.27.2 Member Function Documentation . . . 117
7.27.2.1 send . . . 117
7.28 prism::TX_POSIX_unicast_port Class Reference . . . 118
7.28.1 Detailed Description . . . 119
7.28.2 Member Function Documentation . . . 119
7.28.2.1 send . . . 119
7.29 prism::TX_prism_port Class Reference . . . 120

7.29.1 Detailed Description 121
7.29.2 Constructor & Destructor Documentation 121
7.29.2.1 TX_prism_port 121
7.29.3 Member Function Documentation 122
7.29.3.1 send 122

List of Figures

1.1 A high-level system architecture of a networked embedded system 18

2.1 Lockstep synchronization in PALS (PRISM) 21
2.2 Ideal Harmonic synchronization demonstrated in PRISM . . . 22
2.3 Client-server synchronization in PALS/PRISM 23
2.4 Ideal intra-period synchronization demonstrated by PALS/PRISM 25
2.5 Environment input synchronizer in the PALS system (lockstep synchronization) 26
2.6 Environment output synchronizer in the PALS system (lockstep synchronization) 26

3.1 Internal architecture of a synchronization task 30

4.1 Class diagram of PRISM system 36
4.2 Class diagram for system dependent part for communications . 37
4.3 Class diagram for system dependent part for threads 38
4.4 Activities between `Prism_task` class and its subclasses 40
4.5 Sequence and activity diagrams of `TX_prism_port` and `Port_buffer` 41
4.6 Sequence diagram for `RX_prism_port` and `Port_buffer` . . . 42
4.7 Activity diagram of `RX_prism_port` for packet reception . . . 43
4.8 Sequence diagram between tasks in pure PALS semantics and client-server PALS semantics 44

Version History

Version 0.51

- Bug fixes found by *Coverity*. There was uninitialized member variables in `Realtime_fprintf`.
- `Prism_task::prism_state`, a deprecated member variable was removed and neat `Prism_task::run()` member function follows.
- `DBG_PRINTF` can be used with any prism header.
- Some prologues of the files were corrected.
- Some JSF code convention was corrected.
- `LICENSE.txt` file includes the LMC license.

Version 0.50

- Bug fixes from *CodeSonar*. There had been a bug that accesses the memory after releasing the packet buffer when `RX_prism_port::recv_port_buffer()` removes stale packets from the queue that have missed its own timers for deliveries.
- Code stabilization in `RX_prism_port::recv_port_buffer()`. To prohibit, memory overflow caused by excessive packet queueing, the library limits the packet queueing time and the number of queued packets in `RX_prism_port`. The limits are specified at the constructors

of `RX_prism_port`. For more detail, read the API references for the constructors.

- `POSIX_RX_port` must specify the size of receiving buffer when receiving packets using the `recv_port_buffer` member function. See the API reference manual.

Version 0.49

- System dependent code of threads and timers are decoupled from the `Prism_task` class. To use a thread system other than `pthread`, the developers will implement a subclass of `Abstract_thread_engine` for the new thread system.
- Some APIs have changes for better JSF compliance
  - Namespace `prism` is employed for all the PRISM related code. JSF coding standards regulates that all the nonlocal names should be placed in some namespace. "`using namespace prism;`" is advised to be added at the header of all C++ files.
  - Changes: `Prism_manager` class is added for better object-orientation.
    * `prism_manager.h` should be included when `Prism_manager` class is used.
    * `initialize_prism()` function is transferred to `Prism_manager` class. At the call, (1) thread factory is specified, and (2) the use of `rt_fprintf()` is determined. If the users wants to display the debug messages through `rt_fprintf()`, set it to true. If the users do not want to display messages using the API or have other display mechanisms, set it to false. As a result, the new call is: `Prism_manager::initialize_prism(` *thread factory, use of* `rt_fprintf)`.
  - Changes: `posix_timer.h` was extended to have thread-related class declarations and renamed into `posix_thread.h`. Now, the original `posix_timer.h` of version 0.48 is part of the `posix_thread.h`.
  - Changes: `posix` or `Posix` used in any name is changed to `POSIX` since JSF coding standard regulates that acronyms will be composed of uppercase letters.

- Changes: Constants changed to lowercase for JSF coding standard.
- Changes: System time was formerly acquired by `gettimeofday()` and `timeval_to_timespec` has been used for time type casting. Since they are system dependent, it was replaced by `Prism_manager::get_current_time()`, which uses the member function of the default thread factory for PRISM.
- Changes: To optimize buffer copy overhead, `recv_port_buffer(Port_buffer*buf)` was changed to `Port_buffer* recv_port_buffer()`. The caller of the function must release the memory of the returned `Port_buffer` object.
- `Prism_task::wait_completion()` is renamed by `Prism_task::wait_for_completion()`. To make it a better member function name.

Version 0.48

- A thread can set multiple offset timers to divide a PALS period into multiple computation–communication phases with some time offsets. It provides implementation flexibility beyond client-server PALS.
- It decouples the `RX_prism_port` and `TX_prism_port` from UDP/IP. By implementing subclasses of `RX_port` and `TX_port` without a concern of PRISM, PRISM can use other underlying network layers for communications.

Version 0.3

- UDP/IP multicast was added.

Version 0.2

- Computations and communications within a period can be blended in a piece of code. The implementation overhead of PALS protocol is

delegated to the `RX_prism_port` and `TX_prism_port` classes. It also fixes some memory leak problems in the first version.

Version 0.1

- Initial implementation. The initial implementation was based upon the original PALS pattern. It needs explicit division between computations and communications with time intervals to provide the PALS protocol. The users have to compute the scheduling jitter, minimum response time, and minimum network delay explicitly.

Part I

Software Design Description

Chapter 1

Overview

1.1 Introduction

This Software Design Description (SDD) provides detailed information on the architecture and coding for the PRISM C++ library (version 0.49). The PRISM C++ library supports consistent information sharing and interactions between distributed components of networked embedded systems, e.g. avionics. It is designed to reduce the complexity of the networked system by employing synchronous semantics provided by the architectural pattern called a *Physically-Asynchronous Logically-Synchronous (PALS)* system [1, 3, 4, 5].

Networked embedded systems consist of a network of nodes which are driven by distributed clocks with drifts and errors. While the clock errors of different nodes can be bounded, they cannot be completely eliminated. As a result, when interactions between these nodes are directly driven by their local clocks, the resulting interactions become asynchronous. In the aviation community, this architecture is commonly known as Globally Asynchronous Locally Synchronous (GALS) architecture, since the local synchronous computations of distributed nodes execute asynchronously with respect to each other.

Designing and verifying distributed synchronization protocols in this GALS architecture is extremely difficult when distributed components require consistent views, consistent actions and synchronized state transitions in real-

time to guarantee safety of the system.

Under the GALS architecture, asynchronous interactions may lead to distributed race conditions. As a result, different subsystems may operate inconsistently and impact the system safety adversely. To illustrate this problem, let us consider a distributed synchronization protocol which implements the leader election or the active-standby logic for a dual-redundant flight guidance system (FGS). In this system, two replicated flight guidance systems must have consistent views of which side is the leader or the active side. They must also perform consistent actions, e.g. switching the active/standby mode upon the pilot command. Suppose that the dual-redundant FGS is built on a GALS architecture and the two clocks of the FGS have a bounded skew of 1 ms with respect to a global clock. As a result, one subsystem can be in state j but the other lags by 2 ms and remains in state $j-1$. Hence a pilot command for a mode change arrives in one system in state j while the other may receive the command in state $j-1$ leading to potential divergence between the replicated machines.

In the aviation community, these race conditions are major contributors of the *No Fault Found*, the No. 1 complaint by airlines [2]. When a reported problem cannot be duplicated during repair service, the box is sent back to the customers to use and the cycle repeats. Tracking down these problems or the source of race conditions is indeed like finding a needle in a haystack. Formal analysis tools, e.g. a model checker, may also fail to find any counter-example within a limited time period, which may even span more than a day. Miller et. al. [4] showed this non-triviality of the formal verification in case of the active-standby design of an avionics application in the GALS architecture. It was found that model checking the asynchronous model took over 35 hours even to discover a counter example, compared to validating a correct synchronous model in less than 30 seconds.

The source of this verification complexity is the *state explosion problem* in a GALS architecture resulting from the exponential growth of the asynchronous interactions. A model checker has to explore all possible state interleaving among different nodes under all possible clock skew combinations tick by tick, creating a combinatorial explosion of the interaction state space.

The complexities of debugging in the GALS system motivated us to solve the GALS problem by a *system architecture design approach*, not by the de-

bugging during development and maintenance. Moreover, correcting a race condition in an ad-hoc manner can easily lead to more unforeseen errors due to the difficulty of comprehending all possible asynchronous interactions. In our previous work, we proposed an architectural pattern, called a Physically Asynchronous Logically Synchronous (PALS) system, that systematically eliminates race conditions arising from the asynchronous interactions. This pattern allows developers to design, verify and implement a logically synchronous design of distributed computations.

In the PALS system design, designers at first design a synchronous solution of the distributed computation assuming that the distributed clocks were perfectly synchronized. Since the synchronous solution is conceptually easy to grasp, designers are less likely to make errors and errors, if made, are easier to be detected and corrected. The PALS pattern systematically reuses this synchronous design and distributes on the physically asynchronous architecture without any changes in the application logic, even though there is no global clock in this architecture. The formal analysis of the correctness and optimality of the PALS pattern can be found at [3, 5]. An Architecture Analysis and Design Language (AADL) model description and pattern validation of the PALS pattern can be found at [1].

We have implemented a library, called PRISM, to implement the PALS pattern on top of a real-time communication system. Any global computations requiring consistent views and consistent actions are executed using this library. It uses the PALS pattern to eliminate distributed clocks related race conditions and allows us to design a distributed, redundant system as if all nodes execute synchronously. Designers can code the application logic in a similar way as a synchronous system. It simplifies the development and verification of distributed applications and ensures optimal system performance.

In addition to the basic PALS design described in [1] allowing single rate of executions in distributed PALS tasks, the PRISM library allows multiple execution rates in the tasks, and multiple offset timers for a task during a PALS period. Such extensions in PRISM provide more flexibility for implementation in practice.

1.1.1 Purpose

The purpose of this document is to help readers understand the motivation, design principle of the PALS pattern and the use of the PRISM for application development.

1.1.2 Scope

The PRISM library is applicable in hard real-time systems, such as avionics, that guarantee bounded end-to-end delay including communication and computation time. The networked system must also support periodic clock synchronization to keep the clock skew bounded. Such clock synchronization can be achieved either in hardware or software. It is also assumed that the clocks are monotonically increasing, *i.e.* clocks cannot be reset to the past. Furthermore, the maximum rate of clock changes during correction should be explicitly stated so that application or system developers can take this information into account. As long as these requirements are satisfied, the PRISM library can essentially reduce the design and verification costs of a complex asynchronous system so that it matches the cost of the simpler synchronous system design.

1.2 System Overview

An example target system is depicted in Figure 1.1. The target networked embedded system of PRISM consists of distributed components with sensors and actuators connected by a real-time, fault-tolerant network. There is no global clock in the system. We assume that the local clocks of the distributed components are periodically synchronized so that the clock errors are bounded. Designers can use any robust off-the-shelf or customized clock synchronization algorithm for this purpose. Clock synchronization is not part of the PRISM library.

The PRISM framework consists of PRISM tasks, each of which has its own period, called PALS period. If multiple distributed PRISM tasks have the same PALS period, their executions are synchronized by PRISM library with PALS pattern. If tasks have harmonic periods, their harmonic execution

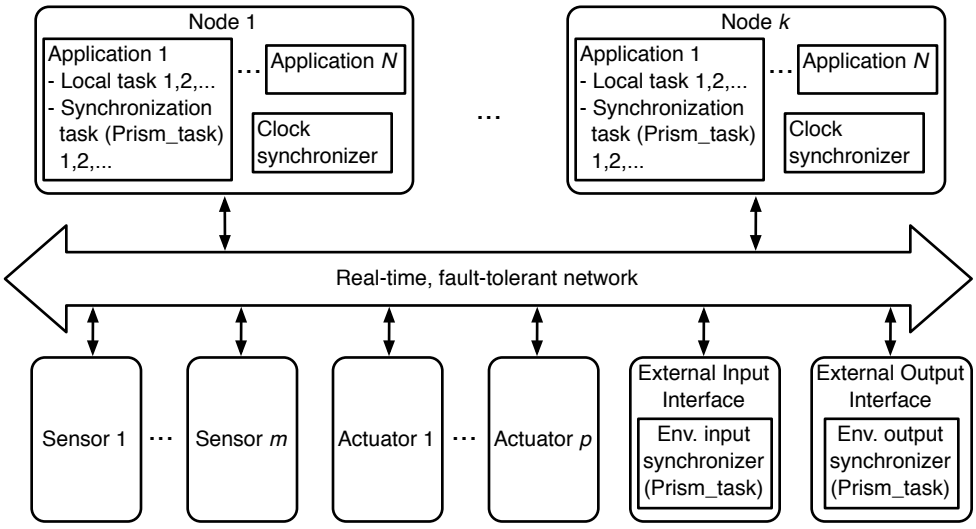


Figure 1.1: A high-level system architecture of a networked embedded system

patterns are also preserved within the framework.

These synchronization tasks are executed using the PRISM library. PRISM defines a base class for periodic threads (defined as `Prism_task` in the library). The developers are expected to extend this base class and define the application logic in the subclass. The message transfers between these tasks are performed through PRISM network interfaces. The PRISM communication interface is defined by the instances of two classes, `RX_prism_port` and `TX_prism_port`, which are used for receiving and transmitting datagram packets. These objects guarantee logically synchronous message deliveries to the destination synchronization tasks even though these tasks are not perfectly synchronized.

The synchronization tasks of distributed nodes may interact in external environment components. For example, the pilots may interact with the flight control system through a flight crew interface. The pilot commands to these synchronization tasks must also arrive consistently to the distributed components. The PALS pattern, as well as PRISM based implementation, requires the system to implement an interface task, also known as *environment input synchronizer*. The environment inputs, *e.g.* pilot commands, are sent to the synchronization tasks through this environment input synchronizer. The environment input synchronizer is also derived from the PRISM defined base

period thread and is executed with same PALS period as other synchronization tasks. The communication between the environment input synchronizer and the synchronization tasks are done through the PRISM communication interface.

Similarly, the outputs of the synchronization tasks at distributed nodes are propagated to other environment tasks through another environment interface task known as environment output synchronizer. The purpose of this task is also to give a consistent view of the synchronization tasks to other external systems.

Chapter 2

PRISM Design and Definition

2.1 System Overview

The goal of the PRISM library is to implement logically synchronous distributed design based on the PALS pattern. The PALS pattern as well as the PRISM library supports four patterns of executions and communications of logically synchronous design: lockstep synchronization, harmonic synchronization, client-server synchronization, and intra-period synchronization. Moreover, for consistent and lockstep interaction with the environment outside of the PALS architecture, we employ environment input and output synchronizers.

2.1.1 Lockstep synchronization

In the lockstep synchronization pattern, the nodes of the synchronous model execute in lockstep manner with the same period. At the beginning of each synchronization round or period, the nodes read messages from their input, process the messages and send to other nodes. Messages generated during synchronization round i are consumed by their destination nodes in synchronization round $i + 1$. This design is described in Figure 2.1(a). All the computations of the tasks (i.e. Synchronization Task1, Synchronization Task2 , Synchronization Task3) in the synchronous design are triggered at the same time without intermediate interaction between tasks, called the computation

cycle. After computation is finished, communications between tasks happen, at the communication cycle. One computation cycle and one communication cycle comprise one round of the synchronous design.

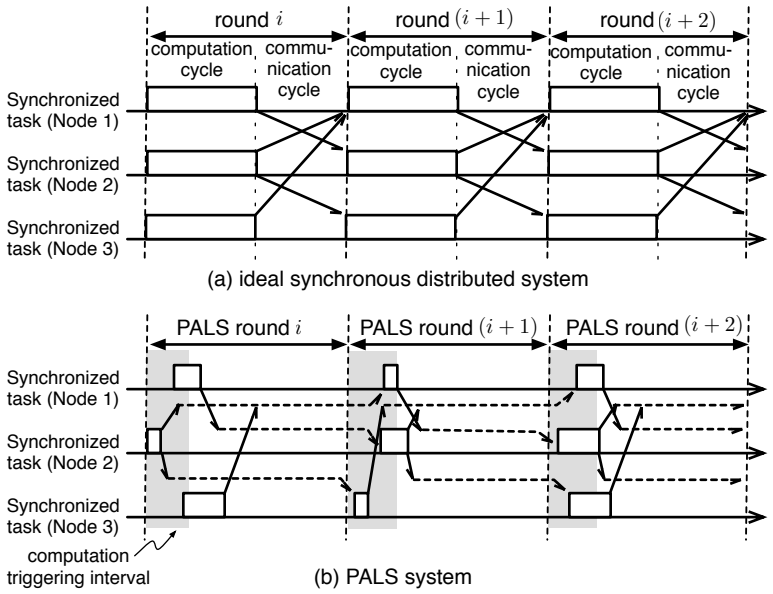


Figure 2.1: Lockstep synchronization in PALS (PRISM)

The PRISM library preserves the same execution and communication behavior of this synchronous design. The basic idea is to trigger distributed computation periodically according to the *PALS period* and only in fixed intervals of time, shadowed area in Figure 2.1(b). Each execution of a PALS period is denoted as the *PALS round*.

The PALS period should be larger than the worst-case end-to-end delay, which is equal to ($2 \times$ maximum clock skew + maximum task response time + maximum network delay). Maximum clock error is defined as the worst-case time difference between a local clock and the ideal global clock at any time. The PRISM library guarantees that messages are processed in the right PALS period according to the synchronous design. Messages generated during PALS round i are consumed by their destination tasks in PALS round $i+1$. Since the clocks are not perfectly synchronized, a message may arrive early at the destination. It may even arrive before the PALS round i of the destination. However, processing at this round would violate the causality and the syn-

chronous semantics. Therefore, the PRISM library appropriately delivers the message in the PALS round $i+1$ at the destination. Thus, the behavior of the distributed system using PRISM library becomes equivalent to that of a synchronous system despite the physical asynchrony in the underlying architecture.

The example code for lockstep synchronization is given at `lockstep_example` directory of the given source code tree with this document. The developers are referred to the code to make lockstep synchronized distributed applications.

2.1.2 Harmonic synchronization

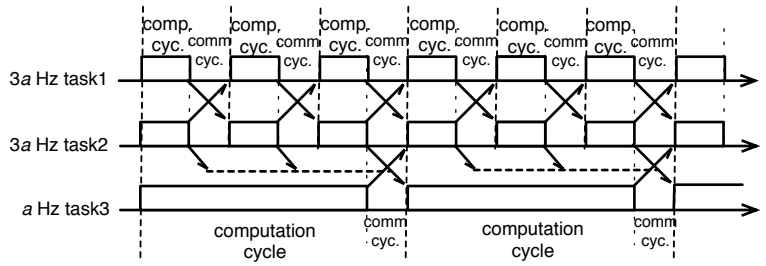


Figure 2.2: Ideal Harmonic synchronization demonstrated in PRISM

In some applications, the execution rates of the distributed tasks are harmonious with each other. Lockstep synchronization can be extended to such applications, too. Fig. 2.2 shows a mixed PALS environment that has lockstep synchronization (between `task1` and `task2`) and harmonic synchronization (between `task2` and `task3`). `task2` has three times higher rate than `task3`. In such cases, three periods of `task2` logically fit into one period of `task3` in PALS/PRISM framework as shown in the figure. In such case, PRISM delivers all the packets issued at the three periods of `task2` to `task3` at the next `task3`'s PALS round as shown in Fig. 2.2. On the other hand, the packets issued by `task3` are delivered to the periods starting at synchronization boundary of the two tasks as shown in the figure.

In theory, the three executions of `task2` is treated as a single execution block for `task3` and every third communication cycle of `task2` is treated as the communication cycle synchronized with `task3`. Such mapping makes a

harmonic synchronization into a lockstep synchronization.

The example code for harmonic synchronization is given in the `harmonic_example` directory of the given source code tree with this document. The developers are referred to the code to make a harmonic synchronized distributed application. Notice that the source code of `harmonic_example` and `lockstep_example` are identical except for task period parameter. PRISM considers a lockstep synchronization as a special case of harmonic synchronization, which the execution of two tasks are one-to-one synchronized, in the implementation.

2.1.3 Client-server synchronization

In the third synchronization pattern, the nodes of the synchronous model execute with the same period. However, the synchronization period is divided into two phases: client phase and server phase. This pattern can be applied in client-server style communication, e.g. sensor-controller-actuator communication where the sensor/actuator is the client, the controller is the server and the actuator receives the output of the controller.

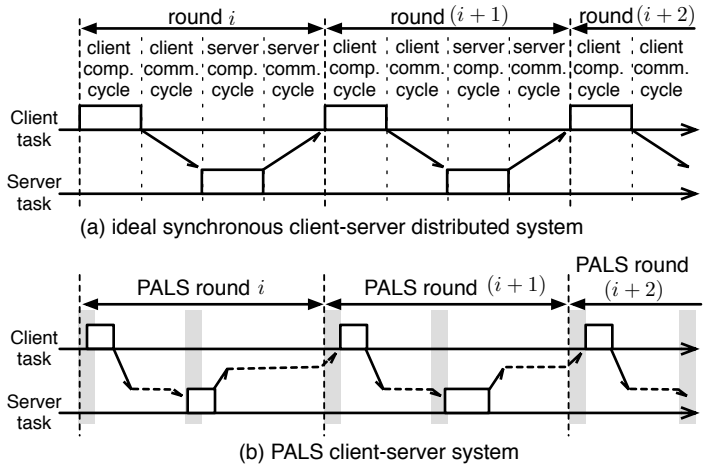


Figure 2.3: Client-server synchronization in PALS/PRISM

In the client phase, the client nodes (e.g. sensor, actuator) begin their computations at the beginning of the synchronization round, process their messages and produce output messages to be sent to the server nodes. At the end of the client phase, the server phase begins. In the server phase, the server

nodes (e.g. controller) process messages from the buffer, perform their computation and send output to the next phase, i.e. the client phase of the next synchronization round. Messages generated during synchronization round i of the client phase are consumed by their destination server tasks in synchronization round i. However, messages generated during synchronization round i of the server phase are consumed by their destination client tasks in synchronization round i+1. The client and server phases comprise one round of the synchronous design. This design is shown in Figure 2.3(a) where we assume that the output of a server task is returned to the same client task.

A networked embedded system using the PRISM library preserves the same execution and communication behavior of a synchronous design with client-server communication. Users of the PRISM library can specify the server and client task and the phase intervals to trigger these distributed computations periodically according to the PALS period with delayed dispatch for the server tasks as shown in Figure 2.3(b). The PALS period should be larger than the worst-case end-to-end delay, which is equal to ($4 \times$ maximum clock skew + maximum client task response time + maximum server task response time + $2 \times$ maximum network delay).

The PRISM library guarantees that messages from the client tasks are received before the beginning of the server phase. Thus messages generated during PALS round i of the client tasks are consumed by their destination server tasks in PALS round i. PRISM also guarantees that server messages are received before the next PALS round at the client tasks. So, messages generated during PALS round i of the server tasks are consumed by their destination client tasks in PALS round $i + 1$.

Although this synchronization can be ideally achieved using the first synchronization of lockstep design, this particular synchronization simplifies the application code since the application does not have to keep track of the phases internally. Moreover, the period of synchronization may be reduced if the worst-case response times of the phases are not same.

The example code for client-server synchronization is given at `client-server-example` directory of the given source code tree with this document. The developers are referred to the code to make a client-server distributed application.

2.1.4 Intra-period synchronization

By a request from users, we made a more general synchronization model than the client-server synchronization, called *intra-period* synchronization. In the intra-period synchronization, a PALS period can be sub divided into multiple execution fragments by employing offset timers. Within a PALS period, developers can set multiple timers having different offsets from the beginning of each period. A packet issued in a intra-period time fragment can explicitly target an offset time fragment of another task in the same PALS round as shown in Fig. 2.4. Meanwhile, some packets can target the next PALS round through lockstep synchronization. The client-server synchronization is implemented as a kind of intra-period synchronization.

The developer is responsible for the feasibility of the intra-period synchronization. PRISM just provides interfaces to realize such synchronizations.

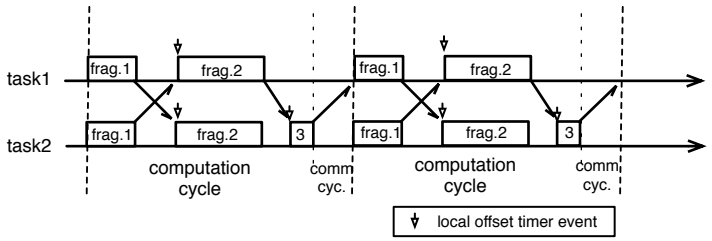


Figure 2.4: Ideal intra-period synchronization demonstrated by PALS/PRISM

The example code for intra-period synchronization is given at `complicated-example` directory of the given source code tree with this document. The developers are referred to the code to make a intra-period distributed application.

2.1.5 Environment input/output synchronizer

The role of the environment input synchronizer is to propagate the external inputs consistently to the synchronization tasks. The environment input synchronizer is also implemented using the PRISM library and it is executed at the same rate of PALS period and within a certain bounded error with respect to the global time. Any input received in the PALS round i are

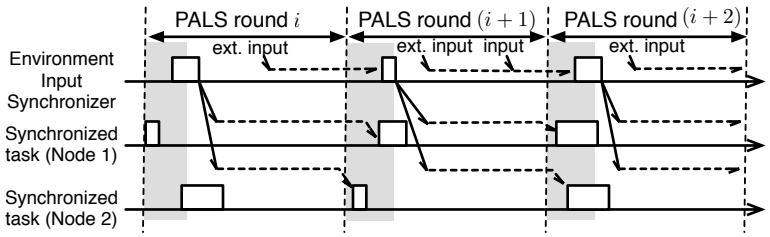


Figure 2.5: Environment input synchronizer in the PALS system (lockstep synchronization)

propagated to the synchronization tasks in the same round and delivered at PALS round $i+1$ as shown in Figure 2.5.

Similarly an environment output synchronizer is used if the outputs of the synchronization tasks are required to be delivered consistently to an external interface. The messages from the PALS round i of the synchronization tasks are propagated to the environment output synchronizer in the same round and delivered at the next round (Figure 2.6).

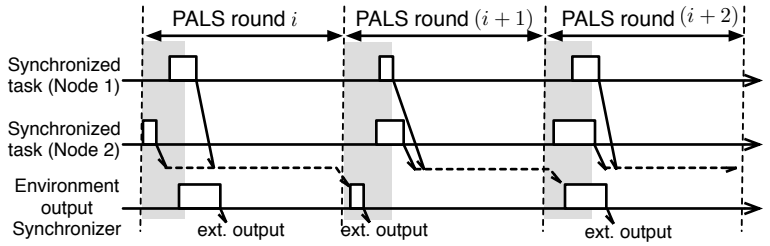


Figure 2.6: Environment output synchronizer in the PALS system (lockstep synchronization)

2.2 Hardware Architecture

We have tested on x86 architecture. However, the library does not have any platform specific implementation code. The message communications are based on the standard network byte order endianness.

The communication architecture must be fault-tolerant and must guarantee reliable message transfer with bounded network delay.

2.3 Support Software

PRISM currently runs on a Linux platform and is dependent upon real-time libraries: pthread and rt library. These libraries are usually installed with a Linux distribution.

Moreover, PRISM is designed to have a good portability. All the system dependent part of the library is decoupled in `/prism/sysdep` directory. Currently, POSIX/UDP is the only supported system, but it can be extended to support other systems by adding appropriate system dependent layer.

2.4 Implementation Details

PRISM is developed in C++. PRISM is implemented as an application library to be used in user-space. In addition to the base classes for logical synchronization, it also includes some support object-oriented libraries for thread implements and network communications.

2.5 Software Scalability and Flexibility

The change in the network topology or distribution of applications may affect the clock skew, network delay and computation time. These effects must be taken into account when computing the PALS period in the PRISM library. However, in these cases, the change in the application code is very minimal. Software designers only have to modify the PALS period parameter for each distributed computation using the PRISM.

A single instance of the PRISM library is used per application process in a node. The PRISM library is flexible to support more than one distributed computation in a process at each node. It also supports all four synchronizations: lockstep synchronization, harmonic synchronization, client-server synchronization, and intra-period synchronization of distributed computations.

The support libraries of PRISM give basic functionalities for marshalling and unmarshalling of these data elements.

PRISM supports multiple timers for a single thread, intra-period synchronization for support of legacy system better and since system dependent code is independent from the core PRISM logic, the system can be easily extended to support non-POSIX systems.

Chapter 3

System Architectural Design

The PRISM library provides the necessary execution and communication interfaces for achieving logical synchronization of the global synchronization tasks. Since the synchronization tasks at different nodes and the environment input and output synchronizers are part of this logically synchronous design, they execute periodically according to the PALS period. Although these tasks cannot be perfectly synchronized because of the local clock skews, the PRISM library guarantees that messages are correctly delivered at the appropriate period depending on the desired synchronization requirement, e.g. lockstep synchronization, harmonic synchronization, client-server synchronization, and intra-period synchronization.

There are three core classes in PRISM library for PALS protocol realization: `Prism_task`, `TX_prism_port` and `RX_prism_port`. The design of a synchronization task inside an application using the PRISM library is given in Figure 3.1. The UML design and internal structure of these classes are discussed in Chapter 4. In this section, we describe the responsibilities of these classes and their usage to aid in the design of logically synchronous systems *i.e.* PALS systems.

If developers wants to use a system other than POSIX or UDP/IP, they also have to be familiar with `Prism_manager` class and the classes defined in `abstract_port.[h,cpp]]`, and `abstract_thread.[h,cpp]`.

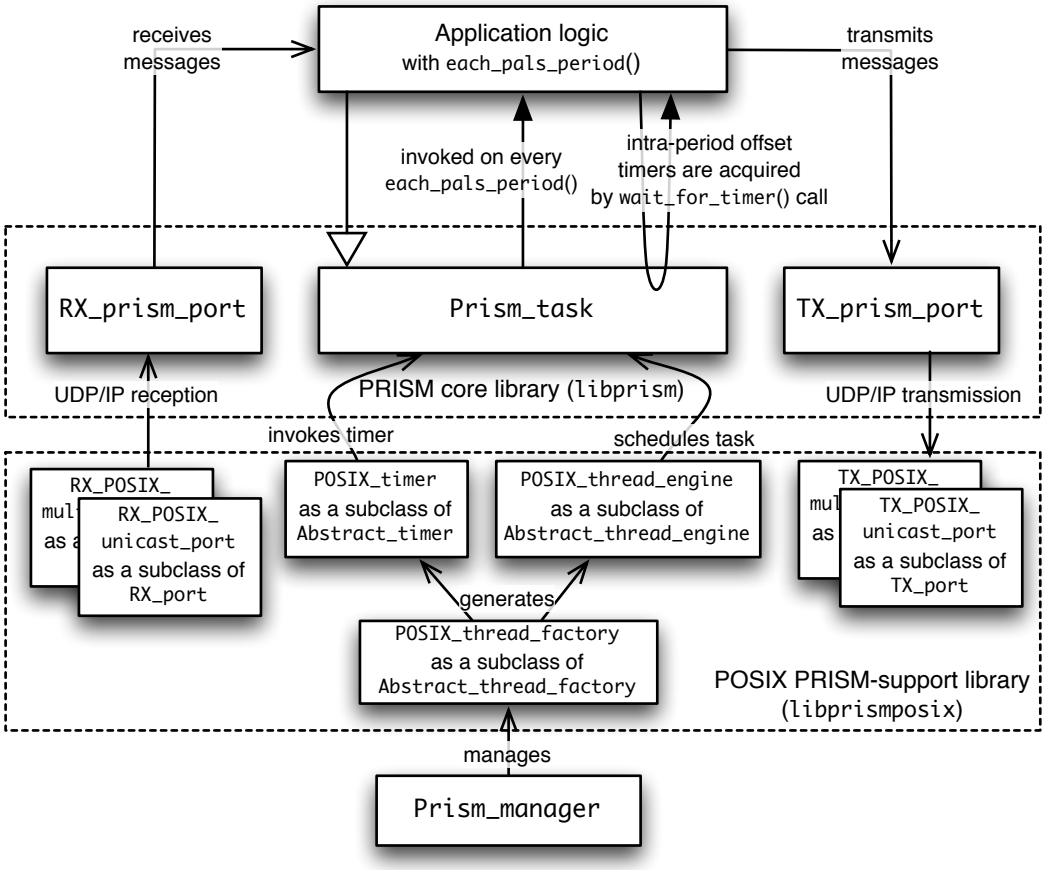


Figure 3.1: Internal architecture of a synchronization task

3.1 System Components

3.1.1 `Prism_manager` and `Abstract_thread_factory`

The PRISM framework a singleton manager class to take care of system dependent part of the system. The most important thing in PRISM is *time* source. All the tasks in the framework must use the dedicated synchronized time sources. Moreover, the thread/timer package for PRISM must use the time source that `Prism_manager` provides. `Abstract_thread_factory` is the interface class for factory classes generates thread engines, locks, condition variables, timers and time sources. On initialization, the application notifies which factory the PRISM is going use, and the factory becomes basis factory creating system-dependent objects for the application. A subclass of the `Abstract_thread_engine` class implements system-dependent parts of the threads. For example, `POSIX_thread_engine` is the subclass to support pthread, and all our examples use the class to use pthread. By implementing a subclass of `Abstract_thread_factory`, we can port PRISM to a new system.

3.1.2 `Prism_task`

The users need to extend the `Prism_task` class for the global synchronization tasks and environment input/output synchronizers to define the application logic for synchronization. The `Prism_task` has a periodic real-time timer (defined as `pals_timer` in the library) with period equal to the PALS period. At each timeout of this timer, it invokes the function `each_pals_period()`. The application logic is defined in this function. In this function, the users extract the messages of the current PALS round or period through a `RX_prism_port` object for each input data. The application transmits data through an object of `TX_prism_port` class of the PRISM library without any delay.

The execution priority of the `Prism_task` and the PALS period length are provided by the user when instantiating the object. Moreover, the user also provides the dispatch offset for the server task when client-server based logical synchronization is employed.

3.1.3 TX_prism_port

TX_prism_port objects are used to transmit messages from a Prism_task object to a destination node. A TX_prism_port object is constructed based on the destination IP address and destination UDP port. Users also have to specify the intended type of synchronization *i.e.* lockstep synchronization, client-server synchronization or general synchronization. A message sent by the TX_prism_port is tagged with the expected deliver time of the destination node, which is calculated based on the current period trigging time and computation fragment timers. RX_prism_port of the destination node can then use this information to deliver the messages at the expected PALS round and computation fragment.

The internals of each message are application dependent. The application logic has to construct the message from the basic data elements. The PRISM library provides functions to create this message conveniently from many data elements of different data sizes. Future versions of the PRISM library are expected to have automatic message construction based on the user specification.

3.1.4 RX_prism_port

RX_prism_port objects are used in the Prism_task to read UDP messages from the network ports. The users use its port_buffer function to read messages for the current PALS round. This object maintains an internal buffer to store any messages that have arrived early and need to be delivered at the right PALS round. When instantiating an RX_prism_port object, the user defines the UDP port and the associated Prism_task reference. The RX_prism_port object uses the reference to the Prism_task to know the current PALS round trigger time and the current computation fragment and uses them to discard any stale messages, store an advanced message or deliver the message of the current PALS round to the application logic.

The PRISM library allows packet transmission without any delay. However, this requires some minor overhead for tagging the PALS round in each message header. The internal logic of the RX_prism_port buffers avoids the causality violation by buffering any early message arrival and delivering at the appropriate period, e.g. the next PALS round when lockstep synchro-

nization is desired.

Any network layer inheriting `TX_port` and `RX_port` can be used as a underlying network layer. As a default, our distribution has `TX_POSIX_unicast_port` and `TX_POSIX_multicast_port` as sample transmitting network layers. `RX_POSIX_unicast_port` and `RX_POSIX_multicast_port` are sample receiving network layers.

3.2 Concept of Execution

The synchronization tasks are executed according to the PALS period required to perform logically synchronous execution. First of all, each task is periodically executed as if it had started its first execution at the time origin, called EPOCH.[1]. Since all the tasks started at the same time, and the clocks are synchronized, their beat of executions are also permanently synchronized as long as their execution rates are the same or harmonic.

If t_{local} denotes the local time, the period boundaries are defined by $t_{\text{local}} \bmod$ `PALS_period` $= 0$. The time of the period starting boundary is called *base time* of a period. If a task is in lockstep synchronization, harmonic synchronization or the a client task of the client-server synchronization task, the timer is triggered at every base time. A `Prism_task` can have multiple timers triggered at a offset shifted time from the base time. A server task of the client-server synchronization defined a offset shift timer, and does not have a timer triggered at a base-time. A task in intra-period synchronization can have multiple timers setup. The offsets of the timers are specified when the PRISM task is constructed.

For performance and neat implementation, the PRISM framework bounds the PALS period to be less than about 4 seconds (`max_pals_period_in_ns` in `prism_task.h`), which is the largest duration in nanoseconds represented by a 32-bit integer.

[1]EPOCH of unix system is January, 1, 1970 00:00

3.3 Interface Design

Please refer to Section 4.

3.4 Software Development Environment

3.4.1 Compilers

PRISM C++ version 0.2 uses g++ and gcc compiler. It has been developed for version 4.4.5 of these compilers. GNU make tool (v 3.81) is also required.

3.4.2 Operating Systems

Ubuntu 9.10 has been used for development and testing.

Chapter 4

PRISM Detailed Design

To present the design, we first introduce the relationship between classes in the class diagram, and introduce use-cases with use-case diagrams followed by other diagrams to describe class interactions.

4.1 Class Diagram for PRISM core

The relationships between classes and the objects represented in Figure 3.1 can also be depicted by Figure 4.1 in UML class diagram. User-defined logic is supposed to be declared as a subclass of `Prism_task`. It must redefine `each_pals_period()` member function, and may redefine `initialize()` if special phase of initialization is needed. If intra-period offset timers are defined for a task, `wait_for_timer()` member function will be called from user-defined `each_pals_period()` member function.

Communications for a PRISM task must be performed through `TX_prism_port` and `RX_prism_port`, which realizes the PALS communication protocol in a system-independent manner. Each `prism_port` object is supposed to serve a single specific PRISM task; it cannot serve multiple tasks. `TX_prism_port` has to specifies the destination in the constructor with the delivery period (the same or the next period) in `port_property` parameter, and the target offset timer of the destination in `target_timer_index` parameter. The constructor of `RX_prism_port` can limit the buffering capacity to prohibit memory overflow through the parameters, `max_periods_for_queue`

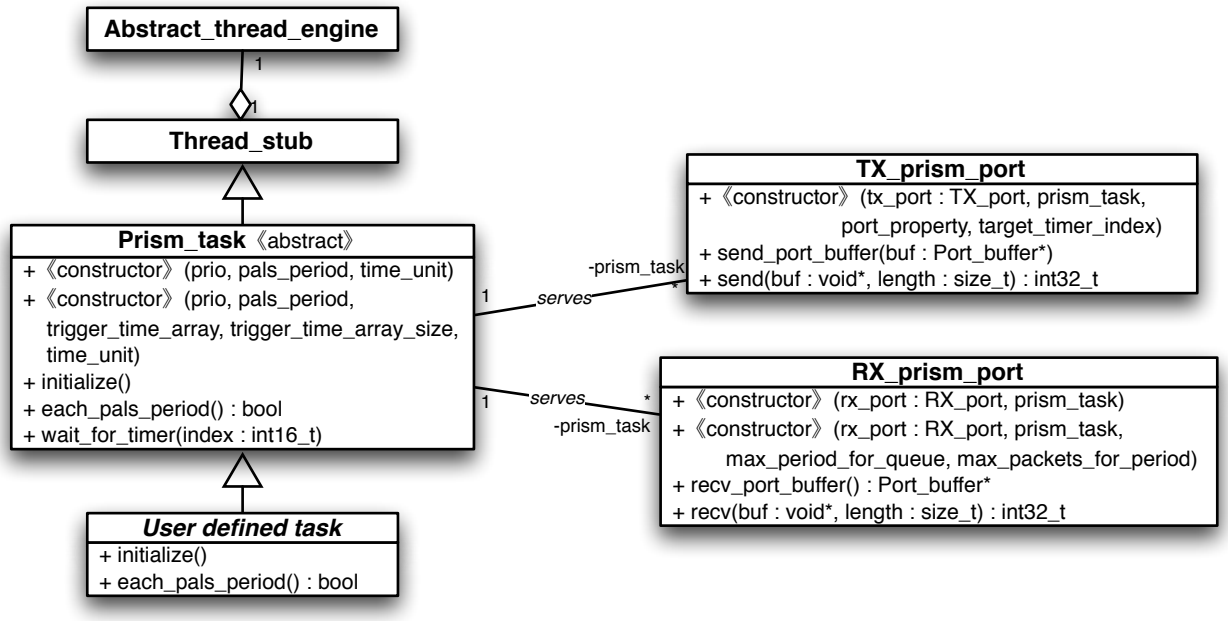


Figure 4.1: Class diagram of PRISM system

and `max_packets_for_period`. The details of the parameters are given in the API reference manual.

4.2 Class Diagram for system dependent part

The POSIX implementation of system dependent part of the system is in `prism/sysdep/posix` directory. It basically implements the interfaces for system-dependent part defined in `abstract_thread.[h,cpp]` and `abstract_port.[h,cpp]` for threads and communications, respectively.

4.2.1 System-dependent communications

The class relationships for system-dependent communications are depicted in Figure 4.2. The lower communication layer defining system-dependent communications is refered by a private variable `port` in `TX_prism_port` and `RX_prism_port`. The object representing the lower layer must be subclasses

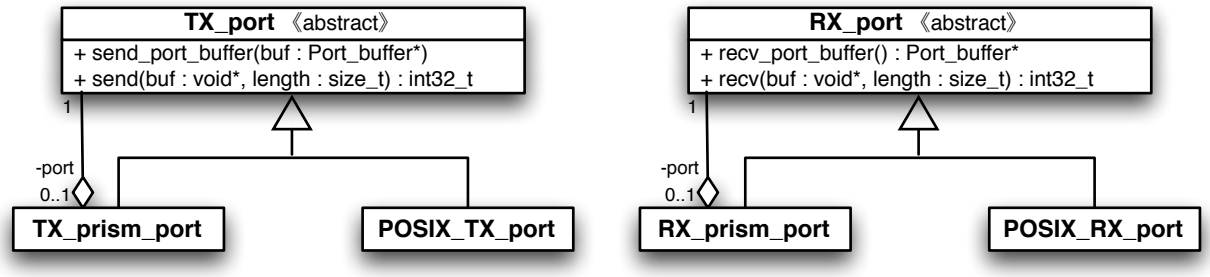


Figure 4.2: Class diagram for system dependent part for communications

of `TX_port` and `RX_port`. Each superclass has two member functions: `send()` and `send_port_buffer()` for `RX_port`, and `recv()` and `recv_port_buffer()` for `RX_port`. The semantics of `send()` and `recv()` are the same as the ones in POSIX APIs with the same names: sending and receiving messages in a byte array. The other member functions, `send_port_buffer()` and `recv_port_buffer()` are for communications using `Port_buffer` objects.[1]

When developers define a new system-dependent communication layer, they can do their work by only redefining `send()` and `recv()` of the superclasses to have the same semantics of POSIX APIs. Exception handling mechanism for the functions are different from POSIX APIs. For the difference, refer `posix_port.cpp`.

4.2.2 System-dependent thread interfaces

Fig. 4.3 depicts how system-dependent part of the system is organized for thread implementation. The system dependent part must define timer, thread, condition variable, lock, and reference time. When `Prism_task` wants to make system-dependent objects (a timer or a thread), a thread factory creates an object. The thread factory is in a factory design pattern, which is in a subclass of `Abstract_thread_factory`. When initializing, the system registers the default factory to `Prism_manager` singleton object. As far as a new factory is not specified in `Prism_task` construction in the parameters, all the system-dependent objects are created by the default factory.

[1] `Port_buffer` class is a specially designed class to make packet marshalling and unmarshalling convenient.

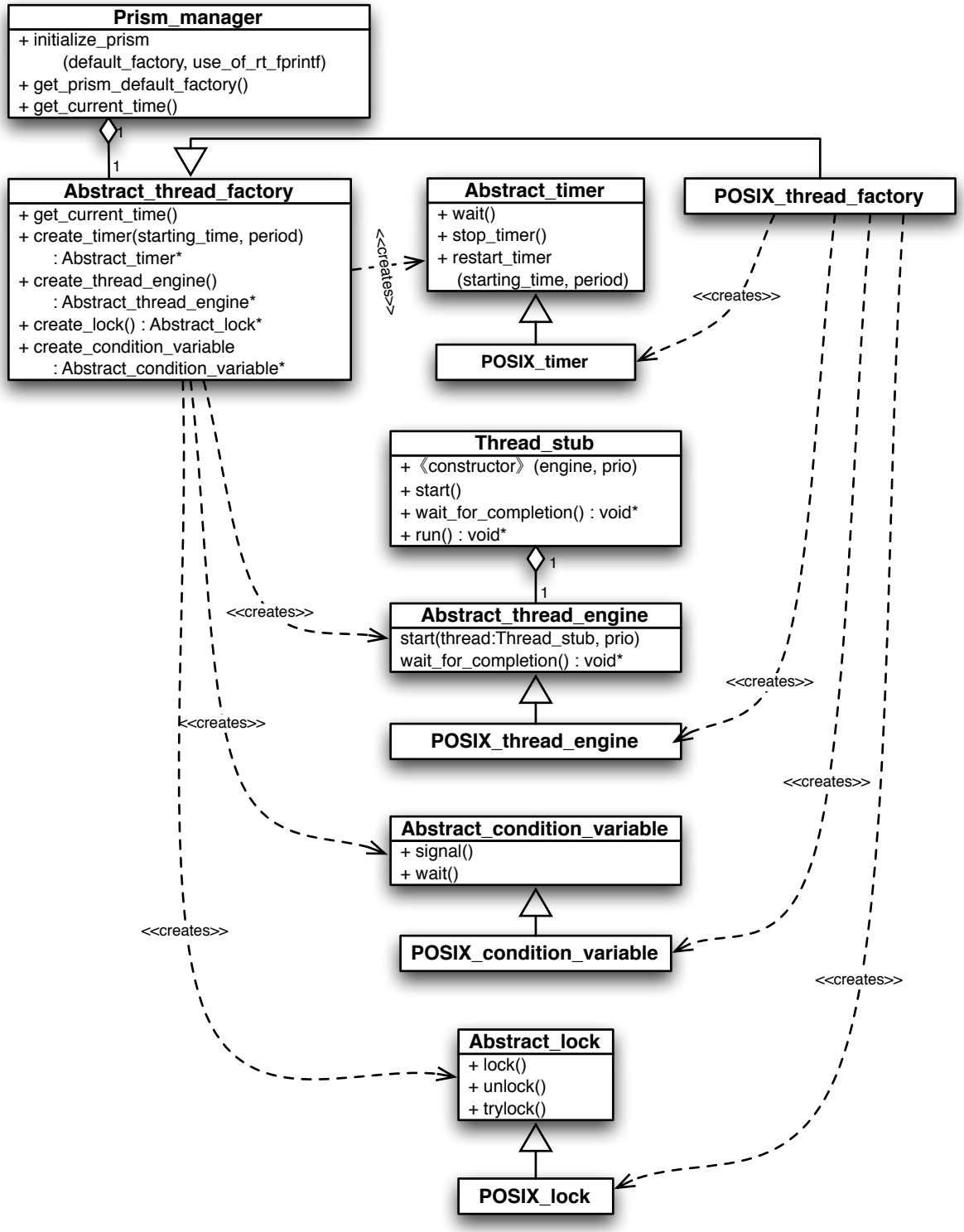


Figure 4.3: Class diagram for system dependent part for threads

4.2.2.0.1 Reference time

The factory also provides the PALS/PRISM reference time. If developers wants to use a PALS/PRISM reference time other than the system time, define a factory class realizing a user defined reference time by redefining `get_current_time()` member function.

4.2.2.0.2 Timers

The semantics of `wait()` and `stop_timer()` are self-explanatory. The semantics of `restart_timer()` is also self-explanatory, but all the queued timers must be cleared in the function.

4.2.2.0.3 Thread engine

To decouple application-specific thread logic from system-dependent thread mechanism, the library has two base classes: `Thread_stub` for application logic, and `Abstract_thread_engine` for system-dependent mechanism. The member functions, `start()` and `wait_for_completion()` of `Abstract_thread_engine` must be redefined by its subclass, and their semantics are self-explanatory.

`Abstract_lock` and `Abstract_condition_variables` are required to be defined accordingly for the thread engine, even though the current PRISM version does not use them.[2]

4.3 Activities between `Prism_task` class and its subclass objects

`Prism_task` is an abstract class that is not supposed to be instantiated by itself. Only its subclasses can be instantiated. An activity diagram showing the interaction between `Prism_task` and its subclass is given in Figure 4.4. The inheriting task defines the initialization routine and the periodic routine by implementing `initialize()` and `each_pals_period()` methods. The

[2]However, `rt_fprintf` functionality uses locks and condition variables.

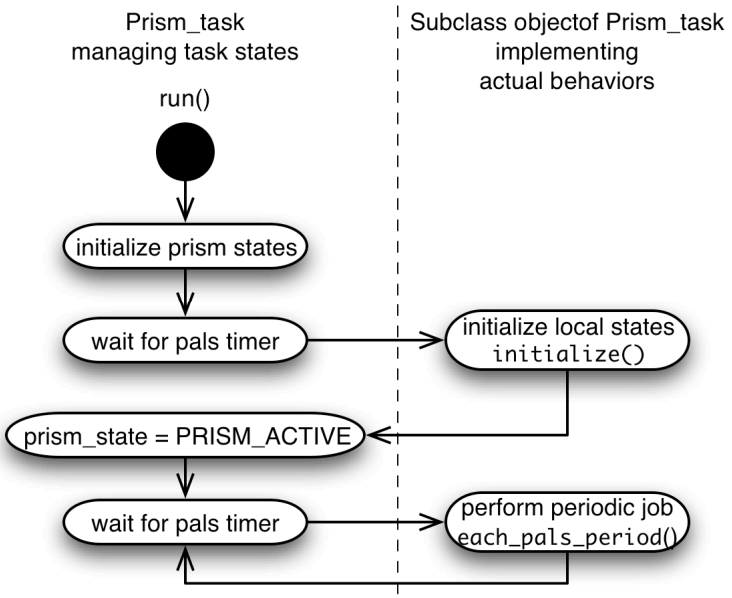


Figure 4.4: Activities between `Prism_task` class and its subclasses

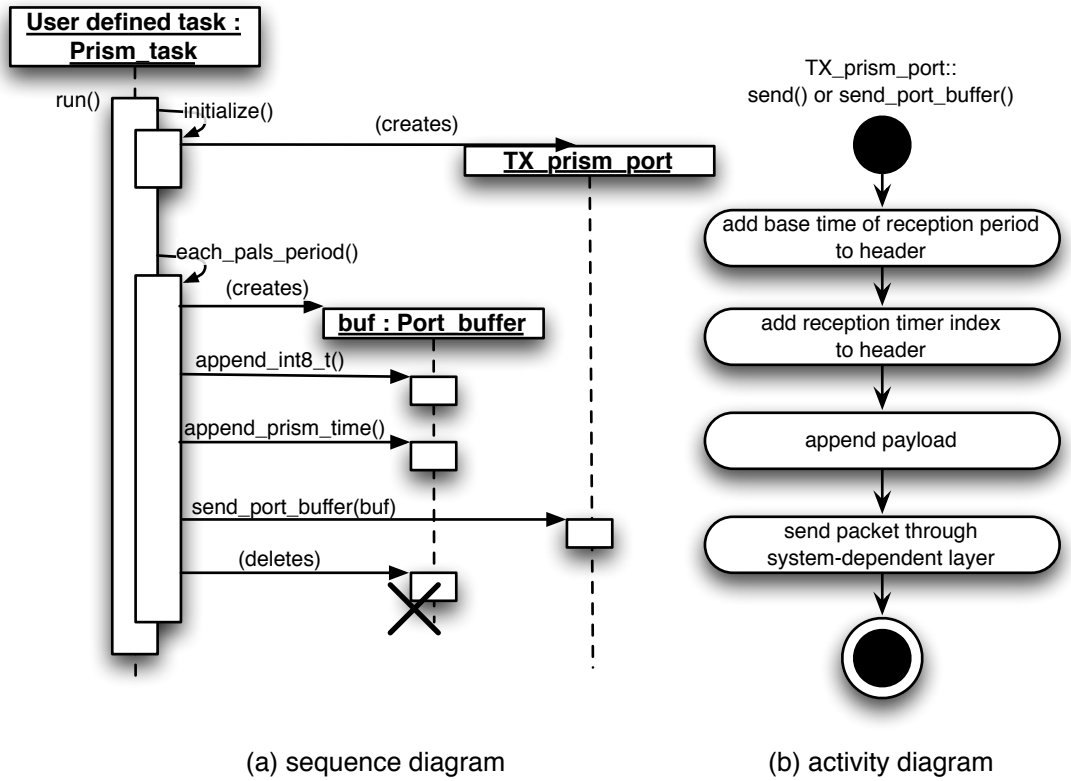
`run()` method, which is the thread main function, invokes the method at the appropriate points of execution.

4.4 Interactions between `Prism_task` and `TX_prism_port`

The inheriting task defines the initialization routine and the periodic routine by implementing `initialize()` and `each_pals_period()` methods. The `run()` method, which is the thread main function, invokes the method at the appropriate points of execution.

`TX_prism_port` sends a packet to be delivered to another and attaches the packet with the expected delivery time. `RX_prism_port` receives the packet and forwards the packet to the task at the appropriate time. Thereby, the complexity is in `RX_prism_port` rather than `TX_prism_port`.

`Prism_buffer` is a buffer class for convenience to deliver primitive data types effectively without concern about the bit endianness. It has a rich set to add primitive data types into the packet buffer. It contains the payload of a packet delivered by `RX_prism_port` and `TX_prism_port`.



(a) sequence diagram (b) activity diagram

Figure 4.5: Sequence and activity diagrams of `TX_prism_port` and `Port_buffer`

4.5 Class and Use Cases for `Prism_task` and `RX_prism_port`

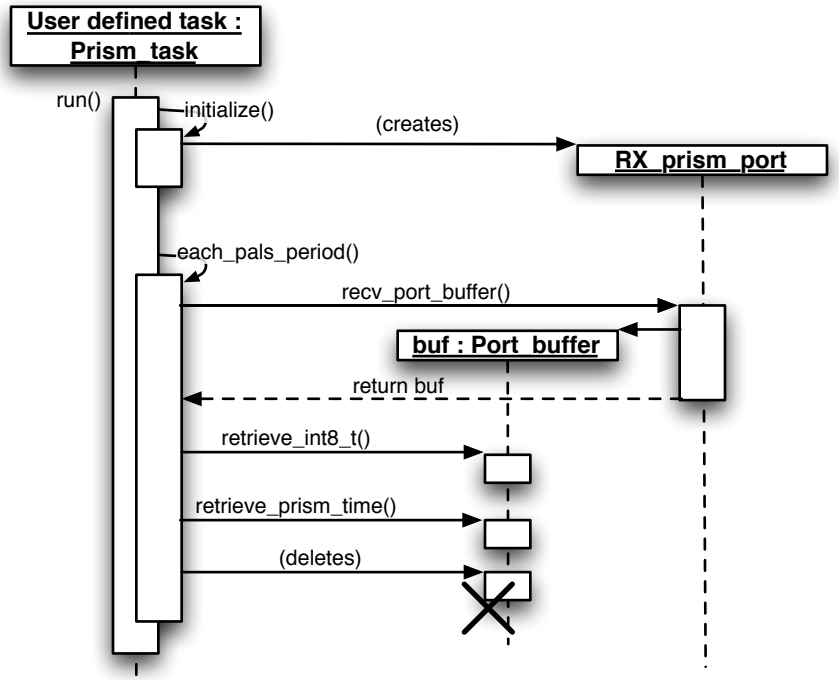


Figure 4.6: Sequence diagram for `RX_prism_port` and `Port_buffer`

The sequence diagram of a packet reception is given in Figure 4.6 which is quite similar to that of packet transmission. However, recall that the internal behavior of `RX_prism_port` is more complicated because the PALS protocol is mostly taken care of by the receiving protocol stack. The activity diagram briefly describing the packet reception activity is given in Figure 4.7. `RX_prism_port` periodically receives the packets and check the expected delivery time. If the current round is later than the expected delivery time, the old packet is discarded. If the packet should be delivered in the current round, `RX_prism_port` will check the header and process the payload. If the packet is for the future round, it will be stored in the `packet_container` and checked again in the future.

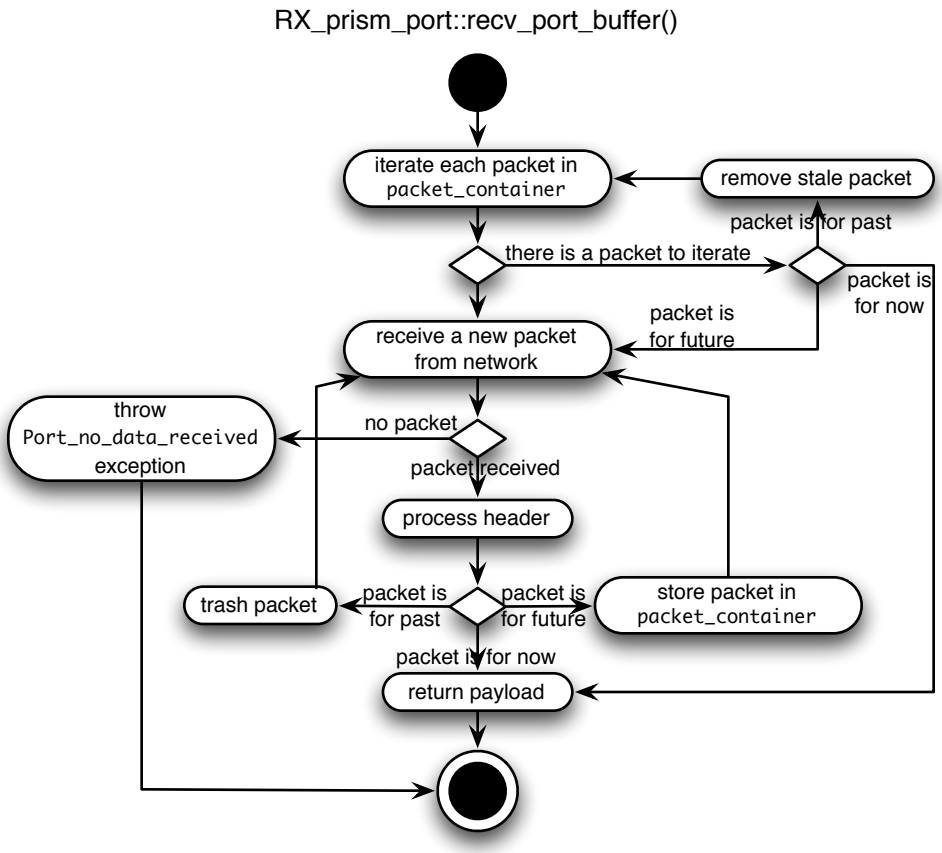
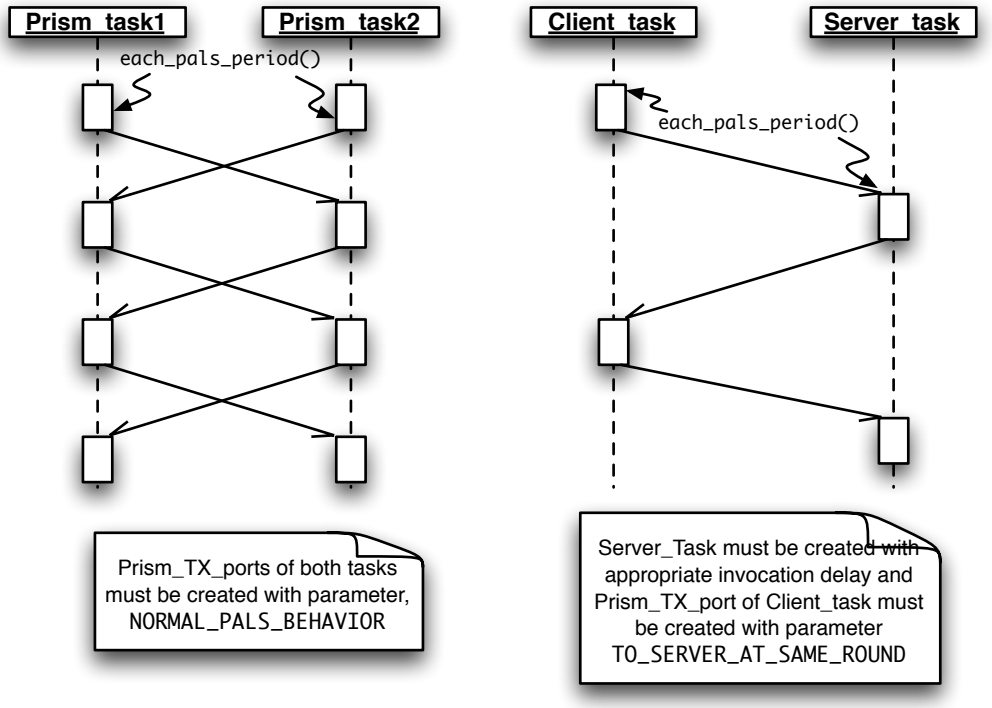


Figure 4.7: Activity diagram of `RX_prism_port` for packet reception

4.6 Pure PALS and Client-server semantics

Recall that the current PRISM supports two kinds of message passing semantics. The semantics can be chosen by using appropriate constructor parameters. The sequence diagram for both semantics are given in Figure 4.8.



(a) sequence diagram for pure PALS (b) sequence diagram for client-server PALS

Figure 4.8: Sequence diagram between tasks in pure PALS semantics and client-server PALS semantics

To employ normal PALS behavior, the subclasses of `PRISM_task` must have zero (0) in the constructor. Moreover, `PRISM_TX_port` must be created with `normal_pals_behavior` for `port_type`.

To employ client-server PALS behavior, the server side of `PRISM_task` must have a positive `trigger_time_array_arg` to wait for the client requests in the constructor. Moreover, `PRISM_TX_port` on the client side must be created with `to_server_at_same_round` for `port_type`.

Acronyms and Abbreviations

SDD	Software Design Description
PALS	Physically-Asynchronous Logically-Synchronous
GALS	Globally-Asynchronous Locally-Synchronous
AADL	Architecture Analysis & Design Language
FGS	Flight Guidance System
GNU	GNU is Not Unix
PRISM	PALS Replicated Interface for Synchronous Modularity

Bibliography

[1] Abdullah Al-Nayeem, Mu Sun, Xiaokang Qiu, Lui Sha, Steven P. Miller, and Darren D. Cofer. A formal architecture pattern for real-time distributed systems. In *Proc. of IEEE RTSS*, 2009.

[2] Bill Burchell. Untangling No Fault Found. *Aviation week – Overhaul & Maintenance*, February 2007.

[3] Jose Meseguer and Peter Olveczky. Formalization and correctness of the pals pattern for asynchronous real-time systems. Technical report, UIUC, 2009.

[4] Steven P. Miller, Darren D. Cofer, Lui Sha, Jose Meseguer, and Abdullah Al-Nayeem. Implementing logical synchrony in integrated modular avionics. In *Proc. of IEEE DASC*, Oct. 2009.

[5] Lui Sha, Abdullah Al-Nayeem, Mu Sun, Jose Meseguer, and Peter Olveczky. Pals: Physically asynchronous logically synchronous systems. Technical report, UIUC, 2009.

Part II

Reference Manual

Chapter 5

Class Index

5.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

prism::Abstract_condition_variable 52
  prism::POSIX_condition_variable 70
prism::Abstract_lock 54
  prism::POSIX_lock 72
prism::Abstract_thread_engine 55
  prism::POSIX_thread_engine 75
prism::Abstract_thread_factory 58
  prism::POSIX_thread_factory 77
prism::Abstract_timer 61
  prism::POSIX_timer 81
prism::Lock_failed 63
prism::Port_buffer 63
prism::Port_buffer_overflow 69
prism::Port_exception 69
prism::Port_no_data_received 70
Print_buffer 84
prism::Prism_manager 85
prism::Prism_time 93

prism::RX_port 98
prism::RX_POSIX_multicast_port 101
prism::RX_POSIX_unicast_port 103
prism::RX_prism_port 106
prism::Thread_stub 110
prism::Prism_task 87
Realtime_fprintf 96
prism::TX_port 113
prism::TX_POSIX_multicast_port 116
prism::TX_POSIX_unicast_port 118
prism::TX_prism_port 120

Chapter 6

Class Index

6.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

prism::Abstract_condition_variable (Condition variable interface to have system-dependent implementations as subclasses) . . 52
prism::Abstract_lock (Lock interface to have system-dependent lock implementations as subclasses) 54
prism::Abstract_thread_engine (Interface class to define system-dependent thread implementation) 55
prism::Abstract_thread_factory (Interface for a factory class to generate system-dependent objects related to threads) 58
prism::Abstract_timer (Interface class to define a timer class that is system-dependent) 61
prism::Lock_failed (Exception triggered by #Abstract_lock::trylock() if it is already locked) 63
prism::Port_buffer (Buffer class supporting appending and retrieving of primitive data types) 63
prism::Port_buffer_overflow (Occurs when a new data is requested to be appended over the buffer size) 69
prism::Port_exception (General network exception) 69
prism::Port_no_data_received (Exception for the case there is no packet received) 70

prism::POSIX_condition_variable (Defines POSIX condition variable implementations) 70
prism::POSIX_lock (Defines posix lock implementation) 72
prism::POSIX_thread_engine (Class supporting a real-time thread in POSIX) 75
prism::POSIX_thread_factory (POSIX Thread factory theat creates thread engines, locks and condition variables) 77
prism::POSIX_timer (Class supporting absolute-time-based real-time timers in POSIX) 81
Print_buffer (Buffer to store the strings to print) 84
prism::Prism_manager (Singleton class managing PRISM framework) 85
prism::Prism_task (Defines periodic prism task as a thread) 87
prism::Prism_time (Class contains time information in nanosecond precision) 93
Realtime_fprintf (Class implementing rt_fprintf mechanism) . . . 96
prism::RX_port (Abstract class to define network port to receive packets) 98
prism::RX_POSIX_multicast_port (POSIX UDP multicast port for packet receptions) 101
prism::RX_POSIX_unicast_port (POSIX UDP unicast port for packet receptions) 103
prism::RX_prism_port (RX communication port in PRISM semantics) 106
prism::Thread_stub (Stub class to define the behavior of a thread) 110
prism::TX_port (Abstract class to define network port to send packets) 113
prism::TX_POSIX_multicast_port (POSIX UDP multicast port for packet transmissions) 116
prism::TX_POSIX_unicast_port (POSIX UDP unicast port for packet transmissions) 118
prism::TX_prism_port (TX communication port in PRISM semantics) 120

Chapter 7

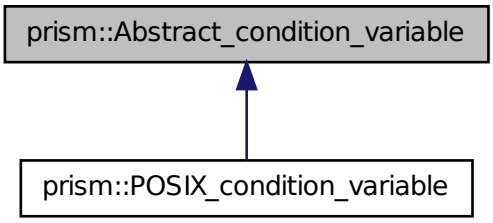
Class Documentation

7.1 prism::Abstract_condition_variable Class Reference

Condition variable interface to have system-dependent implementations as subclasses.

`#include <abstract_thread.h>`

Inheritance diagram for prism::Abstract_condition_variable:



Public Member Functions

- virtual void signal (void)=0

  Sends a signal to a waiting thread.

- virtual void wait (void)=0

  Waits for a signal. Returns when another thread sends the signal.

7.1.1 Detailed Description

Condition variable interface to have system-dependent implementations as subclasses.

7.1.2 Member Function Documentation

7.1.2.1 virtual void prism::Abstract_condition_variable::signal (void) [pure virtual]

Sends a signal to a waiting thread.

Must be defined in the system-dependent subclass.

Implemented in prism::POSIX_condition_variable.

7.1.2.2 virtual void prism::Abstract_condition_variable::wait (void) [pure virtual]

Waits for a signal. Returns when another thread sends the signal.

Must be defined in the system-dependent subclass.

Implemented in prism::POSIX_condition_variable.

The documentation for this class was generated from the following file:

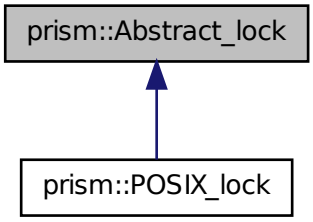
- prism/include/abstract_thread.h

7.2 prism::Abstract_lock Class Reference

Lock interface to have system-dependent lock implementations as subclasses.

`#include <abstract_thread.h>`

Inheritance diagram for prism::Abstract_lock:



Public Member Functions

- virtual void lock (void)=0

  Aquires the lock.

- virtual void unlock (void)=0

  Releases the lock.

- virtual void trylock (void)=0 throw (Lock_failed)

  Tries the lock. If the lock is free, acquires it. Otherwise, throws Lock_failed exception.

7.2.1 Detailed Description

Lock interface to have system-dependent lock implementations as subclasses.

7.2.2 Member Function Documentation

7.2.2.1 virtual void prism::Abstract_lock::lock (void) [pure virtual]

Aquires the lock.

Must be defined in the system-dependent subclass.

Implemented in prism::POSIX_lock.

7.2.2.2 virtual void prism::Abstract_lock::trylock (void) throw (Lock_failed) [pure virtual]

Tries the lock. If the lock is free, acquires it. Otherwise, throws *Lock_failed* exception.

Must be defined in the system-dependent subclass.

Implemented in prism::POSIX_lock.

7.2.2.3 virtual void prism::Abstract_lock::unlock (void) [pure virtual]

Releases the lock.

Must be defined in the system-dependent subclass.

Implemented in prism::POSIX_lock.

The documentation for this class was generated from the following file:

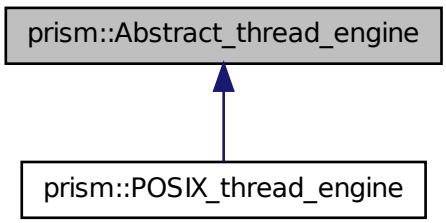
- prism/include/abstract_thread.h

7.3 prism::Abstract_thread_engine Class Reference

Interface class to define system-dependent thread implementation.

`#include <abstract_thread.h>`

Inheritance diagram for prism::Abstract_thread_engine:



Protected Member Functions

- virtual void start (Thread_stub *thread, int16_t priority)=0

  Starts the thread defined by #thread object.

- virtual void * wait_for_completion (Thread_stub *thread)=0

  Returns when the thread completes.

Friends

- class **Thread_stub**

7.3.1 Detailed Description

Interface class to define system-dependent thread implementation. Its subclass must define *how* the thread is realized in the system

7.3.2 Member Function Documentation

7.3.2.1 virtual void prism::Abstract_thread_engine::start (Thread_stub * *thread*, int16_t *priority*) [protected, pure virtual]

Starts the thread defined by #thread object.

Must be defined in the system-dependent subclass.

Parameters

thread	Object that defines thread body
priority	Priority of the thread

Implemented in prism::POSIX_thread_engine.

7.3.2.2 virtual void* prism::Abstract_thread_engine::wait_for_completion (Thread_stub * *thread*) [protected, pure virtual]

Returns when the thread completes.

Must be defined in the system-dependent subclass.

Parameters

thread	Object that defines thread body, the same as the parameter used in start() function

Returns

the pointer value returned by the thread.

Implemented in prism::POSIX_thread_engine.

The documentation for this class was generated from the following file:

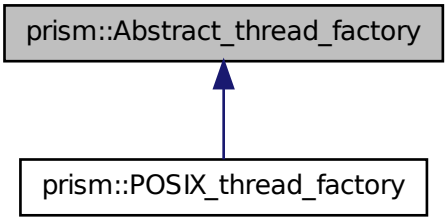
- prism/include/abstract_thread.h

7.4 prism::Abstract_thread_factory Class Reference

Interface for a factory class to generate system-dependent objects related to threads.

`#include <abstract_thread.h>`

Inheritance diagram for prism::Abstract_thread_factory:



Public Member Functions

- virtual Prism_time get_current_time ()=0

  Returns current time.

- virtual Abstract_timer * create_timer (Prism_time &starting_time, Prism_time &period)=0

  Creates a timer.

- virtual Abstract_lock * create_lock ()=0

  Creates a lock.

- virtual Abstract_condition_variable * create_condition_variable (Abstract_lock *lock)=0

  Creates a condition variable.

- virtual Abstract_thread_engine * create_thread_engine ()=0

  Creates a thread engine.

7.4.1 Detailed Description

Interface for a factory class to generate system-dependent objects related to threads.

7.4.2 Member Function Documentation

7.4.2.1 virtual Abstract_condition_variable* prism::Abstract_thread_factory::create_condition_variable (Abstract_lock * *lock*) [pure virtual]

Creates a condition variable.

Must be defined in the system-dependent subclass.

Parameters

lock	condition variable is supposed to be coupled with a lock.

Implemented in prism::POSIX_thread_factory.

7.4.2.2 virtual Abstract_lock* prism::Abstract_thread_factory::create_lock ( ) [pure virtual]

Creates a lock.

Must be defined in the system-dependent subclass.

Implemented in prism::POSIX_thread_factory.

7.4.2.3 virtual Abstract_thread_engine* prism::Abstract_thread_factory::create_thread_engine ( ) [pure virtual]

Creates a thread engine.

Must be defined in the system-dependent subclass.

Implemented in prism::POSIX_thread_factory.

7.4.2.4 virtual Abstract_timer* prism::Abstract_thread_factory::create_timer (Prism_time & *starting_time*, Prism_time & *period*) [pure virtual]

Creates a timer.

Must be defined in the system-dependent subclass.

Implemented in prism::POSIX_thread_factory.

7.4.2.5 virtual Prism_time prism::Abstract_thread_factory::get_current_time ( ) [pure virtual]

Returns current time.

Since different system APIs can provide different time, Prism needs to have a unified time source. The default thread factory for #Prism_manager generates the standard time.

Returns

current time

Implemented in prism::POSIX_thread_factory.

The documentation for this class was generated from the following file:

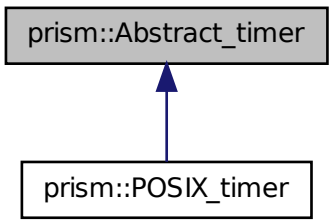
- prism/include/abstract_thread.h

7.5 prism::Abstract_timer Class Reference

Interface class to define a timer class that is system-dependent.

`#include <abstract_thread.h>`

Inheritance diagram for prism::Abstract_timer:



Public Member Functions

- virtual void wait ()=0

  Waits for the timer event.

- virtual void stop_timer ()=0

  Stops the timer.

- virtual void restart_timer (Prism_time &starting_time, Prism_time &period)=0

  Resets the timer with the new parameters.

7.5.1 Detailed Description

Interface class to define a timer class that is system-dependent.

7.5.2 Member Function Documentation

7.5.2.1 virtual void prism::Abstract_timer::restart_timer (Prism_time & *starting_time*, Prism_time & *period*) [pure virtual]

Resets the timer with the new parameters.

The old timer events must be cleaned up.

Parameters

starting_time	restarting time for the timer
period	repeating period

Implemented in prism::POSIX_timer.

7.5.2.2 virtual void prism::Abstract_timer::stop_timer ( ) [pure virtual]

Stops the timer.

The derived function must clear all the related timer events (signals).

Implemented in prism::POSIX_timer.

7.5.2.3 virtual void prism::Abstract_timer::wait ( ) [pure virtual]

Waits for the timer event.

If just one timer event has passed, it will immediately return. If more than two timers have passed, it will immediately return. However, the number of timer events processed by the return is unspecified as POSIX timer.

Implemented in prism::POSIX_timer.

The documentation for this class was generated from the following file:

- prism/include/abstract_thread.h

7.6 prism::Lock_failed Class Reference

Exception triggered by #Abstract_lock::trylock() if it is already locked.

`#include <abstract_thread.h>`

7.6.1 Detailed Description

Exception triggered by #Abstract_lock::trylock() if it is already locked.

The documentation for this class was generated from the following file:

- prism/include/abstract_thread.h

7.7 prism::Port_buffer Class Reference

Buffer class supporting appending and retrieving of primitive data types.

`#include <port_buffer.h>`

Public Member Functions

- Port_buffer (size_t max)

  Constructs a buffer specifying the maximum size of buffer in byte.

- virtual ~Port_buffer (void)

  Destructs the buffer.

- int32_t size ()

  Returns the current buffer size.

- int32_t get_max_size ()

  Returns allocated buffer size.

- int32_t get_remained_size ()

Returns remained number of bytes in buffer.

- void flush ()

  Removes all the data in buffer.

- virtual char * tostr ()

  Generates printable hexadecimal string from contents of buffer.

- virtual void **append_bool** (bool)
- virtual void **append_uint8** (uint8_t)
- virtual void **append_uint16** (uint16_t)
- virtual void **append_uint32** (uint32_t)
- virtual void **append_uint64** (uint64_t)
- virtual void append_byte_arr (char *arr, int32_t arrlen)

  Appends an array to the current buffer.

- virtual void append_buf (Port_buffer *buf)

  Append a buffer to the current buffer.

- virtual void append_time (Prism_time time)

  Append time information to the current buffer.

- virtual void **append_int8** (int8_t n)
- virtual void **append_int16** (int16_t n)
- virtual void **append_int32** (int32_t n)
- virtual void **append_int64** (int64_t n)
- virtual void **append_uint8_t** (uint8_t n)
- virtual void **append_uint16_t** (uint16_t n)
- virtual void **append_uint32_t** (uint32_t n)
- virtual void **append_uint64_t** (uint64_t n)
- virtual void **append_int8_t** (int8_t n)
- virtual void **append_int16_t** (int16_t n)
- virtual void **append_int32_t** (int32_t n)
- virtual void **append_int64_t** (int64_t n)

- virtual void append_float (float n)
- virtual void append_double (double n)
- virtual bool **retrieve_bool** ()
- virtual uint8_t **retrieve_uint8** ()
- virtual uint16_t **retrieve_uint16** ()
- virtual uint32_t **retrieve_uint32** ()
- virtual uint64_t **retrieve_uint64** ()
- virtual int32_t retrieve_byte_arr (char *arr, int32_t max_len)

  Returns number of bytes copied to the array.

- virtual Prism_time **retrieve_time** ()
- virtual char **retrieve_int8** ()
- virtual int16_t **retrieve_int16** ()
- virtual int32_t **retrieve_int32** ()
- virtual int64_t **retrieve_int64** ()
- virtual uint8_t **retrieve_uint8_t** ()
- virtual int16_t **retrieve_uint16_t** ()
- virtual int32_t **retrieve_uint32_t** ()
- virtual int64_t **retrieve_uint64_t** ()
- virtual char **retrieve_int8_t** ()
- virtual int16_t **retrieve_int16_t** ()
- virtual int32_t **retrieve_int32_t** ()
- virtual int64_t **retrieve_int64_t** ()
- virtual float retrieve_float ()
- virtual double retrieve_double ()

Static Public Attributes

- static const uint16_t **sizeof_bool** = 1
- static const uint16_t **sizeof_int8_t** = 1
- static const uint16_t **sizeof_int16_t** = 2
- static const uint16_t **sizeof_int32_t** = 4
- static const uint16_t **sizeof_int64_t** = 8

- static const uint16_t **sizeof_uint8_t** = 1
- static const uint16_t **sizeof_uint16_t** = 2
- static const uint16_t **sizeof_uint32_t** = 4
- static const uint16_t **sizeof_uint64_t** = 8
- static const uint16_t **sizeof_float** = 4
- static const uint16_t **sizeof_double** = 8
- static const uint16_t **sizeof_prism_time** = 12

Protected Attributes

- size_t **ptr**
- size_t **len**
- size_t **max_buf_size**
- char * **buf**

Friends

- class **RX_port**
- class **TX_port**

7.7.1 Detailed Description

Buffer class supporting appending and retrieving of primitive data types.

7.7.2 Constructor & Destructor Documentation

7.7.2.1 Port_buffer::Port_buffer (size_t *max*)

Constructs a buffer specifying the maximum size of buffer in byte.

Parameters

max	maximum size of buffer

7.7.3 Member Function Documentation

7.7.3.1 virtual void prism::Port_buffer::append_double (double *n*) [inline, virtual]

Warning

this assumes float is stored in IEEE 754 format and follow machine endianness, either big endianness or small endianness. ref: `http://wiki.debian.org/ArmEabiPort#floating_points`

7.7.3.2 virtual void prism::Port_buffer::append_float (float *n*) [inline, virtual]

Warning

this assumes float is stored in IEEE 754 format and follow machine endianness, either big endianness or small endianness. ref: `http://wiki.debian.org/ArmEabiPort#floating_points`

7.7.3.3 int32_t Port_buffer::get_max_size ( )

Returns allocated buffer size.

Returns

allocated buffer size

7.7.3.4 int32_t Port_buffer::get_remained_size ( )

Returns remained number of bytes in buffer.

Returns

remained number of bytes in buffer

7.7.3.5 virtual double prism::Port_buffer::retrieve_double ( ) [inline, virtual]

Warning

this assumes float is stored in IEEE 754 format and follow machine endianness, either big endianness or small endianness. ref: http://wiki.debian.org/ArmEabiPort floating_points

7.7.3.6 virtual float prism::Port_buffer::retrieve_float ( ) [inline, virtual]

Warning

this assumes float is stored in IEEE 754 format and follow machine endianness, either big endianness or small endianness. ref: http://wiki.debian.org/ArmEabiPort floating_points

7.7.3.7 int32_t Port_buffer::size ( )

Returns the current buffer size.

Returns

the current buffer size

7.7.3.8 char * Port_buffer::tostr ( ) [virtual]

Generates printable hexadecimal string from contents of buffer.

Returns

printable string in hexadecimal representation of buffer

The documentation for this class was generated from the following files:

- prism/include/port_buffer.h
- prism/src/port_buffer.cpp

7.8 prism::Port_buffer_overflow Class Reference

Occurs when a new data is requested to be appended over the buffer size.

`#include <port_buffer.h>`

7.8.1 Detailed Description

Occurs when a new data is requested to be appended over the buffer size.

The documentation for this class was generated from the following file:

- prism/include/port_buffer.h

7.9 prism::Port_exception Class Reference

General network exception.

`#include <abstract_port.h>`

Public Member Functions

- Port_exception (int _errnum)

  Constructs the exception.

Public Attributes

- int err

  Error number specifying the type of exception.

7.9.1 Detailed Description

General network exception. The type of exception is specified by error number.

The documentation for this class was generated from the following file:

- prism/include/abstract_port.h

7.10 prism::Port_no_data_received Class Reference

Exception for the case there is no packet received.

`#include <abstract_port.h>`

7.10.1 Detailed Description

Exception for the case there is no packet received.

The documentation for this class was generated from the following file:

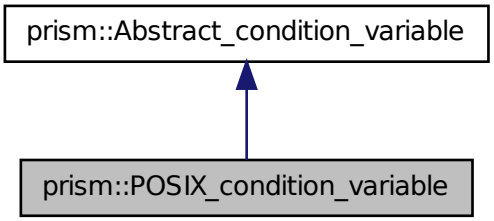
- prism/include/abstract_port.h

7.11 prism::POSIX_condition_variable Class Reference

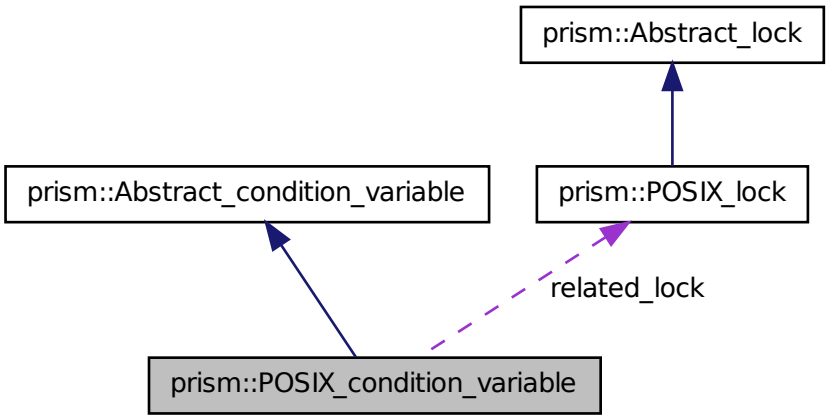
Defines POSIX condition variable implementations.

`#include <posix_thread.h>`

Inheritance diagram for prism::POSIX_condition_variable:



Collaboration diagram for prism::POSIX_condition_variable:



Public Member Functions

- POSIX_condition_variable (POSIX_lock *lock)

  Creates a condition variable.

- ~POSIX_condition_variable ()

  Destroys the condition variable.

- virtual void wait (void)

  Waits for the signal.

- virtual void signal (void)

  Signals the condition variable.

7.11.1 Detailed Description

Defines POSIX condition variable implementations.

The documentation for this class was generated from the following files:

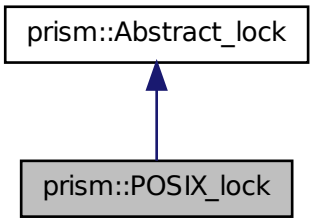
- prism/sysdep/include/posix/posix_thread.h
- prism/sysdep/posix/posix_thread.cpp

7.12 prism::POSIX_lock Class Reference

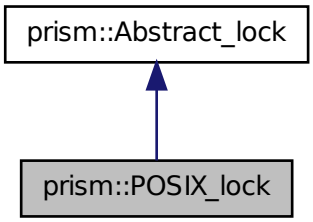
Defines posix lock implementation.

`#include <posix_thread.h>`

Inheritance diagram for prism::POSIX_lock:



Collaboration diagram for prism::POSIX_lock:



Public Member Functions

- POSIX_lock ()
- ~POSIX_lock ()
- virtual void lock (void)
- virtual void unlock (void)
- virtual void trylock (void) throw (Lock_failed)

Friends

- class **POSIX_condition_variable**

7.12.1 Detailed Description

Defines posix lock implementation.

7.12.2 Constructor & Destructor Documentation

7.12.2.1 POSIX_lock::POSIX_lock ( )

Creates a lock

7.12.2.2 POSIX_lock::~POSIX_lock ( )

Destroys the lock

7.12.3 Member Function Documentation

7.12.3.1 void POSIX_lock::lock (void) [virtual]

Aquires the lock.

Implements prism::Abstract_lock.

7.12.3.2 void POSIX_lock::trylock (void) throw (Lock_failed) [virtual]

Tries the lock. If the lock is free, acquires it. Unless, throws #Lock_failed exception.

Implements prism::Abstract_lock.

7.12.3.3 void POSIX_lock::unlock (void) [virtual]

Releases the lock.

Implements prism::Abstract_lock.

The documentation for this class was generated from the following files:

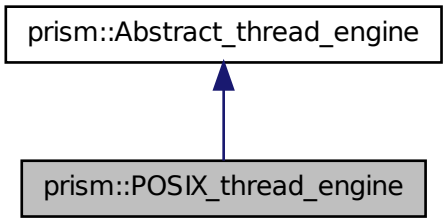
- prism/sysdep/include/posix/posix_thread.h
- prism/sysdep/posix/posix_thread.cpp

7.13 prism::POSIX_thread_engine Class Reference

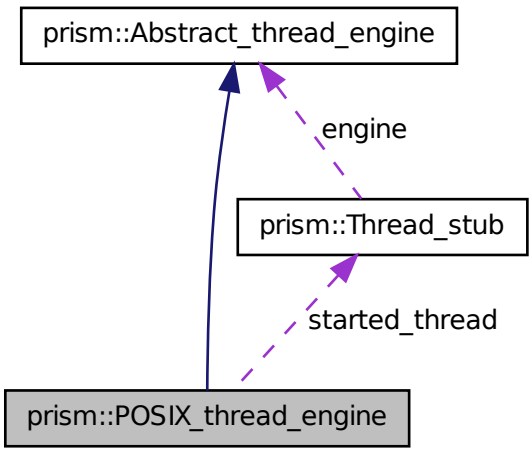
Class supporting a real-time thread in POSIX.

`#include <posix_thread.h>`

Inheritance diagram for prism::POSIX_thread_engine:



Collaboration diagram for prism::POSIX_thread_engine:



Protected Member Functions

- virtual void start (Thread_stub *thread_core, int16_t priority)
- virtual void * wait_for_completion (Thread_stub *thread_core)

7.13.1 Detailed Description

Class supporting a real-time thread in POSIX. The *POSIX_thread_engine* delivers the POSIX thread to the PRISM framework by implementing #Abstract_thread_engine. The thread function is given as a parameter of the start() method.

7.13.2 Member Function Documentation

7.13.2.1 void POSIX_thread_engine::start (Thread_stub * *thread_core*, int16_t *priority*) [protected, virtual]

Starts the thread defined by *thread_function()* in C.

Parameters

thread_function	- function for thread execution.
priority	- priority of the thread.

Implements prism::Abstract_thread_engine.

7.13.2.2 void * POSIX_thread_engine::wait_for_completion (Thread_stub * *thread_core*) [protected, virtual]

Returns when the thread completes.

Returns

the pointer value returned by the thread.

Implements prism::Abstract_thread_engine.

The documentation for this class was generated from the following files:

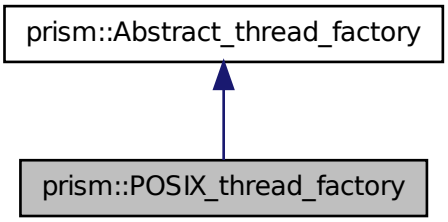
- prism/sysdep/include/posix/posix_thread.h
- prism/sysdep/posix/posix_thread.cpp

7.14 prism::POSIX_thread_factory Class Reference

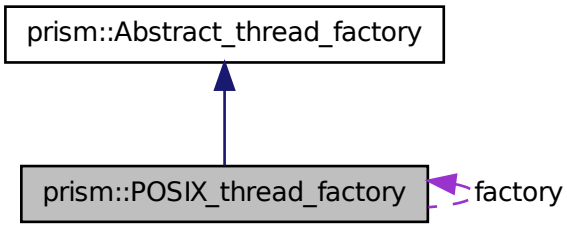
POSIX Thread factory theat creates thread engines, locks and condition variables.

`#include <posix_thread.h>`

Inheritance diagram for prism::POSIX_thread_factory:



Collaboration diagram for prism::POSIX_thread_factory:



Public Member Functions

- virtual Prism_time get_current_time ()

  Returns current time in POSIX API.

- virtual Abstract_timer * create_timer (Prism_time &starting_time, Prism_time &period)

Creates a POSIX timer. Overriden from #Abstract_ thread_ factory::get_ - current_ time().

- virtual Abstract_lock * create_lock ()

  Creates a POSIX thread lock. Overriden from #Abstract_ thread_ factory::create_ - lock().

- virtual Abstract_condition_variable * create_condition_variable (Abstract_ - lock *lock)

  Creates a POSIX thread condition variable.

- virtual Abstract_thread_engine * create_thread_engine ()

  Creates a POSIX thread engine.

Static Public Member Functions

- static POSIX_thread_factory * get_factory ()

  Returns the factory.

7.14.1 Detailed Description

POSIX Thread factory theat creates thread engines, locks and condition variables.

7.14.2 Member Function Documentation

7.14.2.1 Abstract_condition_variable * POSIX_thread_-factory::create_condition_variable (Abstract_lock * *lock*) [virtual]

Creates a POSIX thread condition variable.

Returns

#POSIX_condition_variable object that it created.

Implements prism::Abstract_thread_factory.

7.14.2.2 Abstract_lock * POSIX_thread_factory::create_lock ( ) [virtual]

Creates a POSIX thread lock. Overriden from #Abstract_thread_factory::create_-lock().

Returns

#POSIX_lock object that it created.

Implements prism::Abstract_thread_factory.

7.14.2.3 Abstract_thread_engine * POSIX_thread_factory::create_thread_-engine ( ) [virtual]

Creates a POSIX thread engine.

Returns

#POSIX_thread_engine object that it created.

Implements prism::Abstract_thread_factory.

7.14.2.4 Abstract_timer * POSIX_thread_factory::create_timer (Prism_time & *starting_time*, Prism_time & *period*) [virtual]

Creates a POSIX timer. Overriden from #Abstract_thread_factory::get_-current_time().

Returns

#POSIX_timer object that it created.

Implements prism::Abstract_thread_factory.

7.14.2.5 Prism_time POSIX_thread_factory::get_current_time ( ) [virtual]

Returns current time in POSIX API.

It returns the current time based on *gettimeofday()* POSIX API.

Returns

current time

Implements prism::Abstract_thread_factory.

7.14.2.6 POSIX_thread_factory * POSIX_thread_factory::get_factory ( ) [static]

Returns the factory.

The factory class does not have a state and is a singleton. Therefore, one object can be reused in different places. The object returned by this member function is the one reused. DO NOT try to delete the factory.

The documentation for this class was generated from the following files:

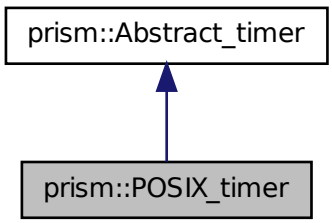
- prism/sysdep/include/posix/posix_thread.h
- prism/sysdep/posix/posix_thread.cpp
- prism/sysdep/posix/posix_timer.cpp

7.15 prism::POSIX_timer Class Reference

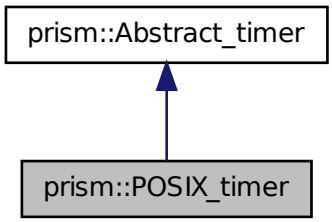
Class supporting absolute-time-based real-time timers in POSIX.

`#include <posix_thread.h>`

Inheritance diagram for prism::POSIX_timer:



Collaboration diagram for prism::POSIX_timer:



Public Member Functions

- POSIX_timer (Prism_time &starting_time, Prism_time &period)

  Constructs a timer with the given starting time and period.

- virtual void wait ()

Waits for the timer event.

- virtual void stop_timer ()

  Stops the timer.

- virtual void restart_timer (Prism_time &starting_time, Prism_time &period)

  Resets the timer with the new parameters.

7.15.1 Detailed Description

Class supporting absolute-time-based real-time timers in POSIX. The POSIX_timer class delivers the POSIX timer to PRISM framework by implementing #Abstract_timer.

See also

Abstract_timer

7.15.2 Constructor & Destructor Documentation

7.15.2.1 POSIX_timer::POSIX_timer (Prism_time & *starting_time*, Prism_time & *period*)

Constructs a timer with the given starting time and period.

period of the timer

Parameters

starting_time	timer starting time

7.15.3 Member Function Documentation

7.15.3.1 void POSIX_timer::restart_timer (Prism_time & *starting_time,* Prism_time & *period*) [virtual]

Resets the timer with the new parameters.

The old timer signals are cleaned up.

Implements prism::Abstract_timer.

7.15.3.2 void POSIX_timer::stop_timer ( ) [virtual]

Stops the timer.

The member function also clears all the related timer events (signals).

< the signal set to wait for the timer, POSIX inter

Implements prism::Abstract_timer.

7.15.3.3 void POSIX_timer::wait (void) [virtual]

Waits for the timer event.

If just one timer event has passed, it will immediately return. If more than two timers have passed, it will immediately return. However, the number of timer events processed by the return is unspecified as POSIX timer. PRISM is designed to work even with this unspecified behavior of a timer.

Implements prism::Abstract_timer.

The documentation for this class was generated from the following files:

- prism/sysdep/include/posix/posix_thread.h
- prism/sysdep/posix/posix_timer.cpp

7.16 Print_buffer Class Reference

Buffer to store the strings to print.

Public Attributes

- FILE * **fd**
- char **buf** [rt_fprintf_buf_size]

7.16.1 Detailed Description

Buffer to store the strings to print.

The documentation for this class was generated from the following file:

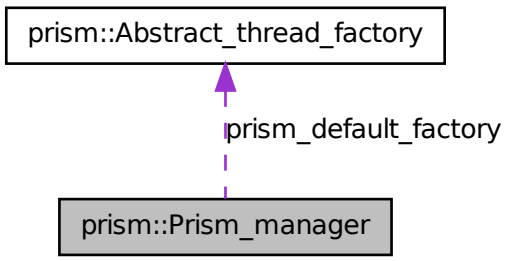
- prism/src/rt_fprintf.cpp

7.17 prism::Prism_manager Class Reference

Singleton class managing PRISM framework.

`#include <prism_manager.h>`

Collaboration diagram for prism::Prism_manager:



Static Public Member Functions

- static void initialize_prism (Abstract_thread_factory *default_factory, bool use_of_rt_fprintf)

  Initializes PRISM with the parameters.

- static Abstract_thread_factory * get_prism_default_factory ()

  Returns the PRISM default factory for thread related operations.

- static bool is_prism_initialized ()

  Returns true if PRISM is initialized.

- static Prism_time get_current_time ()

  Returns the current reference time of PRISM framework, which is the logical global time in PALS.

7.17.1 Detailed Description

Singleton class managing PRISM framework.

7.17.2 Member Function Documentation

7.17.2.1 Prism_time Prism_manager::get_current_time ( ) [static]

Returns the current reference time of PRISM framework, which is the logical global time in PALS.

The system can have multiple time references This function returns the synchronized current time in PRISM semantics

7.17.2.2 Abstract_thread_factory * Prism_manager::get_prism_default_factory ( ) [static]

Returns the PRISM default factory for thread related operations.

Returns

#Abstract_thread_factory object representing the PRISM default factory

7.17.2.3 void Prism_manager::initialize_prism (Abstract_thread_factory * *default_factory*, bool *use_of_rt_fprintf*) [static]

Initializes PRISM with the parameters.

Parameters

default_factory	Default factory for thread related operations
use_of_rt_fprintf	Specifies if #rt_fprintf() should be initialized

7.17.2.4 bool Prism_manager::is_prism_initialized ( ) [static]

Returns true if PRISM is initialized.

Returns

if PRISM is initialized

The documentation for this class was generated from the following files:

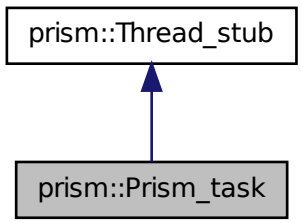
- prism/include/prism_manager.h
- prism/src/prism_manager.cpp

7.18 prism::Prism_task Class Reference

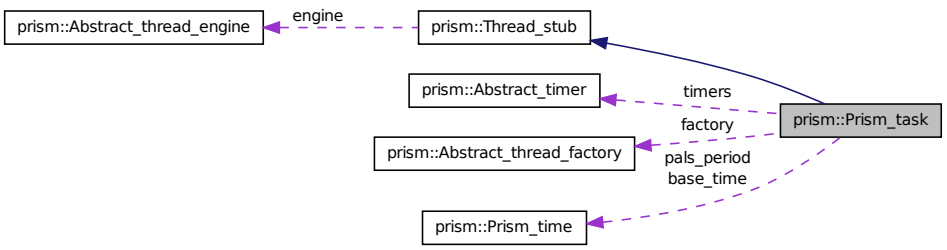
Defines periodic prism task as a thread.

`#include <prism_task.h>`

Inheritance diagram for prism::Prism_task:



Collaboration diagram for prism::Prism_task:



Public Member Functions

- Prism_task (int priority_arg, uint32_t pals_period_arg, uint32_t time_unit)

  Constructs the basic PALS task having no timer offset.

- Prism_task (int priority_arg, uint32_t pals_period_arg, const uint32_t trigger_time_array_arg[], size_t trigger_time_array_size_arg, uint32_t time_unit)

  Constructs the PALS task having a time offset or offsets with the default factory..

- Prism_task (int priority_arg, uint32_t pals_period_arg, const uint32_t trigger_time_array_arg[], size_t trigger_time_array_size_arg, uint32_t time_unit, Abstract_thread_factory *thread_factory)

  Constructs the basic PALS task having a time offset or offsets with the specified factory.

- virtual ~Prism_task ()

  Releases resources for the task.

- Prism_time get_base_time ()

  Gets the base time for PALS round.

- Prism_time get_pals_period ()

  Gets the PALS period.

- uint32_t get_pals_period_in_ns ()

  Gets the PALS period in nanosecond.

- uint16_t get_timer_index ()

  The timer index that just expired.

- void * run (void)

  The main thread function.

Protected Member Functions

- virtual void initialize ()

Called once for initialization at the beginning of thread execution.

- virtual bool each_pals_period (void)

  Main computation function called at every pals period.

- void wait_for_timer (int16_t index)

  Waits for a predefined offset timer defined in task construction.

7.18.1 Detailed Description

Defines periodic prism task as a thread. It has a single period with multiple offset timers. Each offset timer has its own index. Packet deliveries are designated to each timer. timer is initiated as if its base time initially had started at EPOCH.

7.18.2 Constructor & Destructor Documentation

7.18.2.1 Prism_task::Prism_task (int *priority_arg*, uint32_t *pals_period_arg*, uint32_t *time_unit*)

Constructs the basic PALS task having no timer offset.

Default thread factory set at #Prism_manager is used.

7.18.2.2 Prism_task::Prism_task (int *priority_arg*, uint32_t *pals_period_arg*, const uint32_t *trigger_time_array_arg[]*, size_t *trigger_time_array_size_arg*, uint32_t *time_unit*)

Constructs the PALS task having a time offset or offsets with the default factory..

Multiple timers can be triggered for a PALS period for local communications. Default thread factory set at #Prism_manager is used.

7.18.2.3 Prism_task::Prism_task (int *priority_arg*, uint32_t *pals_period_arg*, const uint32_t *trigger_time_array_arg[]*, size_t *trigger_time_array_size_arg*, uint32_t *time_unit*, Abstract_thread_factory * *thread_factory*)

Constructs the basic PALS task having a time offset or offsets with the specified factory.

Multiple timers can be triggered for a PALS period for local communications. This constructor is used when the developer wants to use a thread factory other than the default thread factory specified in #Prism_manager::initialize member function. If you are not sure about the effect of using multiple thread factories in a single process, do not use this constructor.

7.18.3 Member Function Documentation

7.18.3.1 bool Prism_task::each_pals_period (void) [protected, virtual]

Main computation function called at every pals period.

Returns

true if the task does not want to terminate its periodic execution, and *false* if the task wants to terminate its work at this period.

7.18.3.2 Prism_time Prism_task::get_base_time ( )

Gets the base time for PALS round.

Returns

the base time

7.18.3.3 Prism_time Prism_task::get_pals_period ( )

Gets the PALS period.

Returns

the PALS period in #Prism_time

7.18.3.4 uint32_t Prism_task::get_pals_period_in_ns ( )

Gets the PALS period in nanosecond.

Returns

the pals period in nanoseconds

7.18.3.5 uint16_t Prism_task::get_timer_index ( )

The timer index that just expired.

Returns

the current timer's index.

7.18.3.6 void Prism_task::initialize (void) [protected, virtual]

Called once for initialization at the beginning of thread execution.

Called from run() at the beginning and it is synchronized with the PALS period.

7.18.3.7 void * Prism_task::run (void) [virtual]

The main thread function.

Overridden from Thread_stub.

Implements prism::Thread_stub.

7.18.3.8 void Prism_task::wait_for_timer (int16_t *index*) [protected]

Waits for a predefined offset timer defined in task construction.

Called within each_pals_period() to wait for the next timer.

The documentation for this class was generated from the following files:

- prism/include/prism_task.h
- prism/src/prism_task.cpp

7.19 prism::Prism_time Class Reference

Class contains time information in nanosecond precision.

`#include <prism_time.h>`

Public Member Functions

- Prism_time ()
  Constructs a Prism_ time object with origin time (time 0)
- Prism_time (const Prism_time &time)
  Clones a Prism_ time object for a specific time.
- Prism_time (uint64_t second_arg, uint32_t nanosecond_arg)
  Constructs a Prism_ time object for a specific time.
- void set_time (uint64_t second_arg, uint32_t nanosecond_arg)
  Sets the time of the object to the specified one.
- void add_nanoseconds (uint64_t nanoseconds_to_add)
  Adds (unsigned) nanoseconds to the time.
- void add_time (Prism_time &time)
  Adds time.
- void subtract_nanoseconds (uint64_t nanoseconds_to_subtract)
  Subtracts (unsigned) nanoseconds from the time.
- int8_t compare (Prism_time &other_time)
  Compares the time with another Prism_ time object.
- uint64_t get_second ()
  Returns the second.

- uint32_t get_nanosecond ()

  Returns the nanosecond portion.

7.19.1 Detailed Description

Class contains time information in nanosecond precision. It can contain absolute time as well as relative time.

7.19.2 Constructor & Destructor Documentation

7.19.2.1 Prism_time::Prism_time ( )

Constructs a Prism_time object with origin time (time 0)

Origin of absolute time is EPOCH, which can be system dependent. In a single system, EPOCH must be unique.

7.19.3 Member Function Documentation

7.19.3.1 void Prism_time::add_nanoseconds (uint64_t *nanoseconds_to_add*)

Adds (unsigned) nanoseconds to the time.

Parameters

nanoseconds_to_add	nanoseconds to add

7.19.3.2 void Prism_time::add_time (Prism_time & *time*)

Adds time.

Parameters

time	time to add

7.19.3.3 int8_t Prism_time::compare (Prism_time & *other_time*)

Compares the time with another Prism_time object.

Parameters

other_time	object to compare.

Return values

-1	when this is earlier than #other_time,
0	when this is later than #other_time,
1	when two objects have the same time

7.19.3.4 uint32_t prism::Prism_time::get_nanosecond ( ) [inline]

Returns the nanosecond portion.

Returns

nanosecond portion

7.19.3.5 uint64_t prism::Prism_time::get_second ( ) [inline]

Returns the second.

Returns

second

7.19.3.6 void Prism_time::subtract_nanoseconds (uint64_t *nanoseconds_to_subtract*)

Subtracts (unsigned) nanoseconds from the time.

Parameters

nanoseconds_to_subtract	nanoseconds to subtract

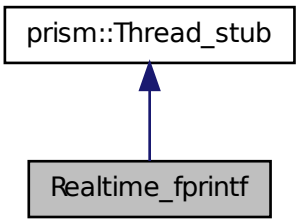
The documentation for this class was generated from the following files:

- prism/include/prism_time.h
- prism/src/prism_time.cpp

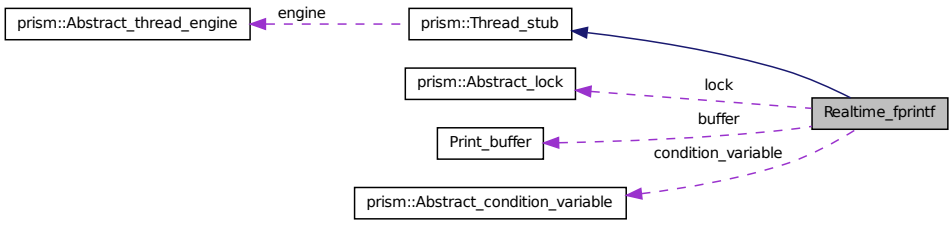
7.20 Realtime_fprintf Class Reference

Class implementing rt_fprintf mechanism.

Inheritance diagram for Realtime_fprintf:



Collaboration diagram for Realtime_fprintf:



Friends

- void initialize_rt_fprintf (Abstract_thread_factory *factory, int16_t priority)

  Before use of # rt_fprintf this function must be called in advance.

- void rt_fprintf (FILE *fd, const char *fmt,...)

  Prints in the same manner as fprintf. But, I/O is taken care of by a low priority task.

7.20.1 Detailed Description

Class implementing rt_fprintf mechanism.

7.20.2 Friends And Related Function Documentation

7.20.2.1 void initialize_rt_fprintf (prism::Abstract_thread_factory * *factory*, int16_t *priority*) [friend]

Before use of # rt_fprintf this function must be called in advance.

Initializes the low-priority thread to take care of I/O

Parameters

factory	Factory creates thread related system objects
priority	I/O thread priority

7.20.2.2 void rt_fprintf (FILE * *fd*, const char * *fmt*, ...) [friend]

Prints in the same manner as fprintf. But, I/O is taken care of by a low priority task.

Usage is the same as *fprintf*

The documentation for this class was generated from the following file:

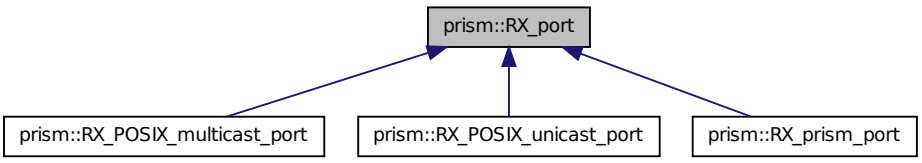
- prism/src/rt_fprintf.cpp

7.21 prism::RX_port Class Reference

Abstract class to define network port to receive packets.

`#include <abstract_port.h>`

Inheritance diagram for prism::RX_port:



Public Member Functions

- RX_port (uint32_t port_buffer_size_arg)

  Constructs RX_port.

- virtual Port_buffer * recv_port_buffer () throw (Port_exception, Port_-no_data_received)

  Receives a packet in a #Port_ buffer object.

- virtual int32_t recv (void *buf, size_t length)=0 throw (Port_exception, Port_no_data_received)

  System dependent implementation of recv goes into this method in the subclass.

- int32_t get_port_buffer_size ()

  Returns buffer size when using #Port_ buffer.

Protected Attributes

- uint32_t **port_buffer_size**

7.21.1 Detailed Description

Abstract class to define network port to receive packets. It provides two packet reception interfaces: reception in raw buffer in *void* * type, and reception in #Port_buffer.

7.21.2 Constructor & Destructor Documentation

7.21.2.1 RX_port::RX_port (uint32_t *port_buffer_size_arg*)

Constructs RX_port.

The subclass defines all the details but the size of #Port_buffer must be commonly specified.

7.21.3 Member Function Documentation

7.21.3.1 virtual int32_t prism::RX_port::recv (void * *buf*, size_t *length*) throw (Port_exception, Port_no_data_received) [pure virtual]

System dependent implementation of *recv* goes into this method in the subclass.

Parameters

buf	the pointer of raw buffer to store the packet
length	the size of #buf

Returns

the number of bytes received

Exceptions

Port_exception	if a network exception is occured.
Port_no_data_received	if there is no packet received.

Implemented in prism::RX_prism_port, prism::RX_POSIX_unicast_port, and prism::RX_POSIX_multicast_port.

7.21.3.2 Port_buffer * RX_port::recv_port_buffer ( ) throw (Port_exception, Port_no_data_received) [virtual]

Receives a packet in a #Port_buffer object.

When the reception is successful, a new object is allocated in heap, and returned. The ownership of the object is transfered to the caller. The caller is responsible for deletion of the returned object.

If a derived class does not override this member function, it simply exploits recv() member function for the operation.

Returns

#Port_buffer object having the received packet

Exceptions

Port_exception	if a network exception is occured.
Port_no_data_received	if there is no packet received.

Reimplemented in prism::RX_prism_port.

The documentation for this class was generated from the following files:

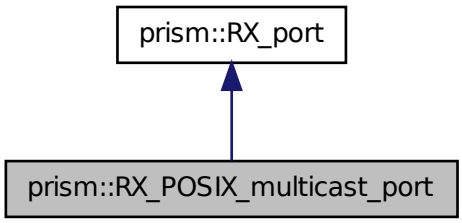
- prism/include/abstract_port.h
- prism/src/abstract_port.cpp

7.22 prism::RX_POSIX_multicast_port Class Reference

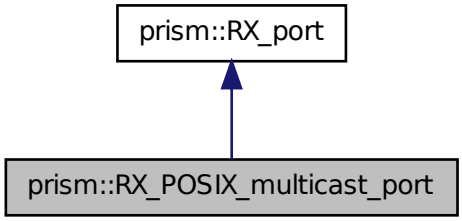
POSIX UDP multicast port for packet receptions.

`#include <posix_port.h>`

Inheritance diagram for prism::RX_POSIX_multicast_port:



Collaboration diagram for prism::RX_POSIX_multicast_port:



Public Member Functions

- RX_POSIX_multicast_port (const char *addr_str, uint16_t port, uint32_t buffer_size_arg)

  Creates POSIX UDP multicast port for packet receptions.

- virtual int32_t recv (void *buf, size_t length) throw (Port_exception, Port_no_data_received)

  POSIX recv in C++.

7.22.1 Detailed Description

POSIX UDP multicast port for packet receptions.

7.22.2 Member Function Documentation

7.22.2.1 int32_t RX_POSIX_multicast_port::recv (void * *buf*, size_t *length*) throw (Port_exception, Port_no_data_received) [virtual]

POSIX *recv* in C++.

Parameters

buf	the pointer of raw buffer to store the packet
length	the size of #buf

Returns

the number of bytes received

Exceptions

Port_exception	if a network exception is occured.
Port_no_data_received	if there is no packet received.

Implements prism::RX_port.

The documentation for this class was generated from the following files:

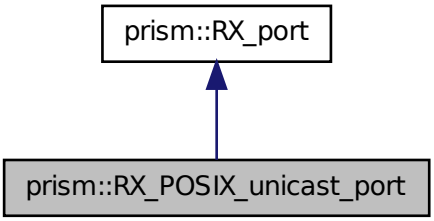
- prism/sysdep/include/posix/posix_port.h
- prism/sysdep/posix/posix_port.cpp

7.23 prism::RX_POSIX_unicast_port Class Reference

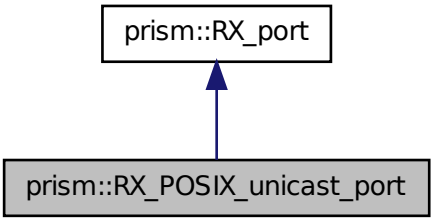
POSIX UDP unicast port for packet receptions.

`#include <posix_port.h>`

Inheritance diagram for prism::RX_POSIX_unicast_port:



Collaboration diagram for prism::RX_POSIX_unicast_port:



Public Member Functions

- RX_POSIX_unicast_port (uint16_t port, uint32_t buffer_size_arg)

  Creates POSIX UDP unicast port for packet receptions.

- virtual int32_t recv (void *buf, size_t length) throw (Port_exception, Port_no_data_received)

  POSIX recv in C++.

7.23.1 Detailed Description

POSIX UDP unicast port for packet receptions.

7.23.2 Member Function Documentation

7.23.2.1 int32_t RX_POSIX_unicast_port::recv (void * *buf*, size_t *length*) throw (Port_exception, Port_no_data_received) [virtual]

POSIX *recv* in C++.

Parameters

buf	the pointer of raw buffer to store the packet
length	the size of #buf

Returns

the number of bytes received

Exceptions

Port_exception	if a network exception is occured.
Port_no_data_received	if there is no packet received.

Implements prism::RX_port.

The documentation for this class was generated from the following files:

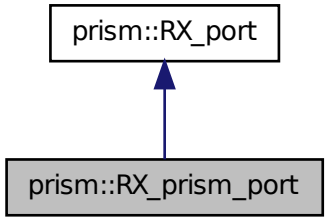
- prism/sysdep/include/posix/posix_port.h
- prism/sysdep/posix/posix_port.cpp

7.24 prism::RX_prism_port Class Reference

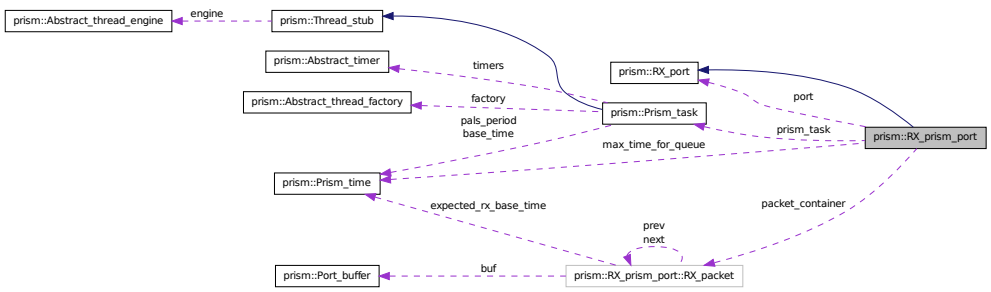
RX communication port in PRISM semantics.

`#include <prism_port.h>`

Inheritance diagram for prism::RX_prism_port:



Collaboration diagram for prism::RX_prism_port:



Classes

- class **RX_packet**

Public Member Functions

- RX_prism_port (RX_port *rx_port_arg, Prism_task *prism_task_arg)

  Constructs a RX_prism_port object.

- RX_prism_port (RX_port *rx_port_arg, Prism_task *prism_task_arg, uint16_t max_periods_for_queue, uint16_t max_packets_per_period)

  Constructs a RX_prism_port object.

- virtual Port_buffer * recv_port_buffer () throw (Port_exception, Port_no_data_received)

  Receives a packet at the expected delivery time in a form of #Port_ buffer object.

- virtual int32_t recv (void *buf, size_t length) throw (Port_exception, Port_no_data_received)

  Receives a packet at the expected delivery time in conventional recv POSIX API manner.

7.24.1 Detailed Description

RX communication port in PRISM semantics. RX_prism_port can be used as a general RX_port, which has interfaces to receive packets. However, it must be always coupled with #TX_prism_port on the other side. RX_prism_port receives a packet, decodes the header and deliver the packet at the time designated by the header.

7.24.2 Constructor & Destructor Documentation

7.24.2.1 RX_prism_port::RX_prism_port (RX_port * *rx_port_arg*, Prism_task * *prism_task_arg*)

Constructs a RX_prism_port object.

Notice that RX_prism_port only defines PRISM semantics not system specific network interfaces. Lower network layer system-specific network interfaces are defined by #rx_port_arg. If the local times of the machines are not well synchronized, A machine with slow time can have too large buffer to keep up other machines' time, and can result in buffer overflow. To detect such situation explicitly, RX_prism_port tracks the base time of a received packet not to be too future. Moreover, it tracks the number of packets in the buffer not to be too large. This constructor does not set such parameters and employ default values: #default_max_packets_in_queue and #default_max_periods_for_queue.

Parameters

rx_port_arg	RX port to receive packets below PRISM layer. The object ownership is transferred to the RX_prism_port object; it is deleted when RX_prism_port object is deleted. Thereby, #rx_port_arg object must be created in heap.
prism_task_arg	#Prism_task using the port.

7.24.2.2 RX_prism_port::RX_prism_port (RX_port * *rx_port_arg*, Prism_task * *prism_task_arg*, uint16_t *max_periods_for_queue*, uint16_t *max_packets_per_period*)

Constructs a RX_prism_port object.

Notice that RX_prism_port only defines PRISM semantics not system specific network interfaces. Lower network layer defining system-specific network interfaces are defined by #rx_port_arg. If the local times of the machines are not well synchronized, A machine with slow time can have too large buffer to keep up other machines' time, and can result in buffer overflow. To detect such situation explicitly, RX_prism_port tracks the base time of a received packet not to be too future. Moreover, it tracks the number of packets in the buffer not to be too large. This constructor explicitly takes the parameters related to these queueing issues.

Parameters

rx_port_arg	RX port to receive packets below PRISM layer. The object ownership is transferred to the RX_prism_port object; it is deleted when RX_prism_port object is deleted. Thereby, #rx_port_arg object must be created in heap.
prism_task_arg	#Prism_task using the port.
max_periods_for_queue	packets cannot be delivered earlier than this period length.
max_packets_per_period	max number of packets per period.

7.24.3 Member Function Documentation

7.24.3.1 int32_t RX_prism_port::recv (void * *buf*, size_t *length*) throw (Port_exception, Port_no_data_received) [virtual]

Receives a packet at the expected delivery time in conventional *recv* POSIX API manner.

Parameters

buf	the pointer of raw buffer to store the packet
length	the size of #buf

Returns

the number of bytes received

Exceptions

Port_exception	if a network exception is occured.
Port_no_data_received	if there is no packet received.

Implements prism::RX_port.

7.24.3.2 Port_buffer * RX_prism_port::recv_port_buffer ( ) throw (Port_exception, Port_no_data_received) [virtual]

Receives a packet at the expected delivery time in a form of #Port_buffer object.

The packet header is examined and decided to be moved to the container for later use or given to the caller in the current period. Note that the returned buffer is in the heap, and ownership of the object is transferred to the caller. The caller must *delete* the buffer after use. The caller need not know about the implementation detail of *packet_container:* If there is a previously received packet in *packet_container*, this function pulls it from the container and returns the pointer of the kept buffer (the caller must free the buffer). Otherwise, this function receives a packet from the lower layer into a freshly allocated buffer (the caller must free the buffer, too).

Returns

the pointer of the #Port_buffer object having the received packet.

Exceptions

Port_exception	if a network exception is occured.
Port_no_data_received	if there is no packet received.

Reimplemented from prism::RX_port.

The documentation for this class was generated from the following files:

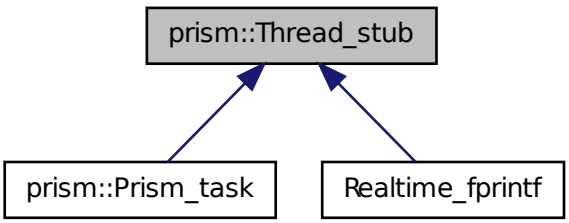
- prism/include/prism_port.h
- prism/src/prism_port.cpp

7.25 prism::Thread_stub Class Reference

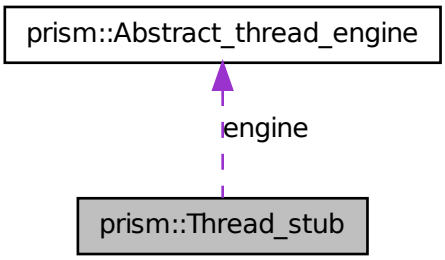
Stub class to define the behavior of a thread.

`#include <abstract_thread.h>`

Inheritance diagram for prism::Thread_stub:



Collaboration diagram for prism::Thread_stub:



Public Member Functions

- Thread_stub (Abstract_thread_engine *engine_arg, int16_t priority_arg)

  Specifies thread-execution parameters.

- void start (void)

  Calls the engine to start thread defined at run() method.

- void * wait_for_completion (void)

  Waits for the thread completion.

- virtual void * run (void)=0

  Represents the thread behavior. Must be defined in the subclass.

7.25.1 Detailed Description

Stub class to define the behavior of a thread. In PRISM, Prism_task is the subclass of this class defining the PRISM thread behavior. System-dependent thread implementation is defined by #Abstract_thread_engine. Hence, the subclass of Thread_stub must not define any system dependent part of the thread.

7.25.2 Constructor & Destructor Documentation

7.25.2.1 Thread_stub::Thread_stub (Abstract_thread_engine * *engine_arg*, int16_t *priority_arg*)

Specifies thread-execution parameters.

Parameters

engine_arg	thread engine defining system dependent thread behavior.
priority_arg	thread priority.

7.25.3 Member Function Documentation

7.25.3.1 virtual void* prism::Thread_stub::run (void) [pure virtual]

Represents the thread behavior. Must be defined in the subclass.

Returns

a pointer value returned at the thread completion.

Implemented in prism::Prism_task.

7.25.3.2 void * Thread_stub::wait_for_completion (void)

Waits for the thread completion.

Engine assists the operation

Returns

a pointer value returned by the thread.

The documentation for this class was generated from the following files:

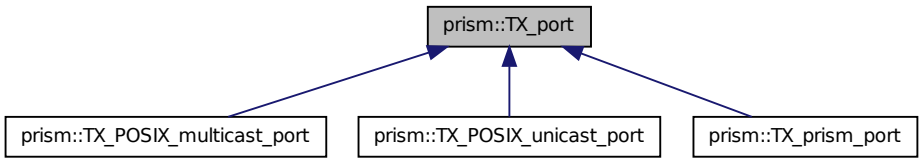
- prism/include/abstract_thread.h
- prism/src/abstract_thread.cpp

7.26 prism::TX_port Class Reference

Abstract class to define network port to send packets.

`#include <abstract_port.h>`

Inheritance diagram for prism::TX_port:



Public Member Functions

- virtual void send_port_buffer (Port_buffer *buf) throw (Port_exception)

  Sends a Port_buffer object as a packet.

- virtual int32_t send (void *buf, size_t length)=0 throw (Port_exception)

  System dependent implementation of send goes into this method.

7.26.1 Detailed Description

Abstract class to define network port to send packets. It provides two packet transmission interfaces: sending a raw buffer in *void* * type, and sending #Port_buffer object.

7.26.2 Member Function Documentation

7.26.2.1 virtual int32_t prism::TX_port::send (void * *buf*, size_t *length*) throw (Port_exception) [pure virtual]

System dependent implementation of *send* goes into this method.

Parameters

buf	the pointer of raw buffer to store the packet
length	the size of #buf

Returns

the number of bytes sent

Exceptions

Port_exception	if a network exception is occured.

Implemented in prism::TX_prism_port, prism::TX_POSIX_unicast_port, and prism::TX_POSIX_multicast_port.

7.26.2.2 void TX_port::send_port_buffer (Port_buffer * *buf*) throw (Port_exception) [virtual]

Sends a Port_buffer object as a packet.

If a derived class does not override this member function, it simply exploits send() member function for the operation.

Parameters

buf	#Port_buffer object to send

Exceptions

Port_exception	if a network exception is occured.

The documentation for this class was generated from the following files:

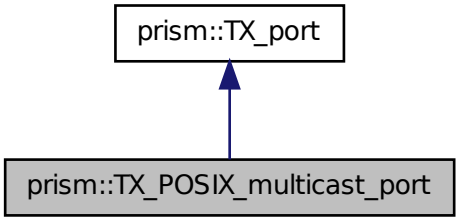
- prism/include/abstract_port.h
- prism/src/abstract_port.cpp

7.27 prism::TX_POSIX_multicast_port Class Reference

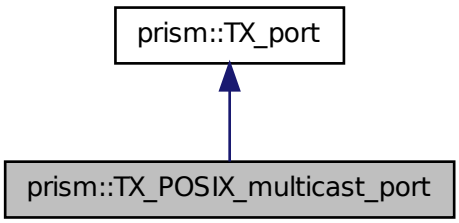
POSIX UDP multicast port for packet transmissions.

`#include <posix_port.h>`

Inheritance diagram for prism::TX_POSIX_multicast_port:



Collaboration diagram for prism::TX_POSIX_multicast_port:



Public Member Functions

- TX_POSIX_multicast_port (const char *addr_str, uint16_t port, int32_t ttl)

  Creates POSIX UDP multicast port for packet transmissions.

- virtual int32_t send (void *buf, size_t length) throw (Port_exception)

  POSIX send in C++.

7.27.1 Detailed Description

POSIX UDP multicast port for packet transmissions.

7.27.2 Member Function Documentation

7.27.2.1 int32_t TX_POSIX_multicast_port::send (void * *buf*, size_t *length*) throw (Port_exception) [virtual]

POSIX *send* in C++.

Parameters

buf	the pointer of raw buffer to store the packet
length	the size of #buf

Returns

the number of bytes sent

Exceptions

Port_exception	if a network exception is occured.

Implements prism::TX_port.

The documentation for this class was generated from the following files:

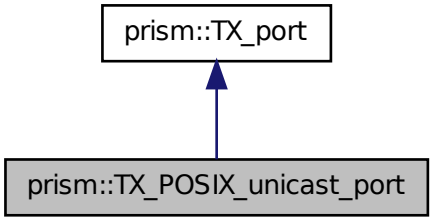
- prism/sysdep/include/posix/posix_port.h
- prism/sysdep/posix/posix_port.cpp

7.28 prism::TX_POSIX_unicast_port Class Reference

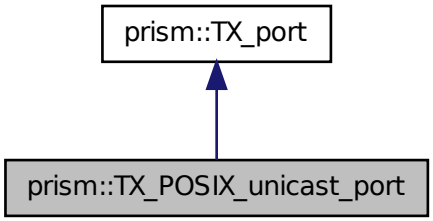
POSIX UDP unicast port for packet transmissions.

`#include <posix_port.h>`

Inheritance diagram for prism::TX_POSIX_unicast_port:



Collaboration diagram for prism::TX_POSIX_unicast_port:



Public Member Functions

- TX_POSIX_unicast_port (const char *addr_str, uint16_t port)
  Creates POSIX UDP unicast port for packet transmissions.
- virtual int32_t send (void *buf, size_t length) throw (Port_exception)

  POSIX send in C++.

7.28.1 Detailed Description

POSIX UDP unicast port for packet transmissions.

7.28.2 Member Function Documentation

7.28.2.1 int32_t TX_POSIX_unicast_port::send (void * *buf*, size_t *length*) throw (Port_exception) [virtual]

POSIX *send* in C++.

Parameters

buf	the pointer of raw buffer to store the packet
length	the size of #buf

Returns

the number of bytes sent

Exceptions

Port_exception	if a network exception is occured.

Implements prism::TX_port.

The documentation for this class was generated from the following files:

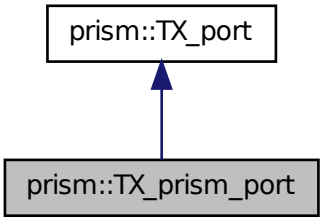
- prism/sysdep/include/posix/posix_port.h
- prism/sysdep/posix/posix_port.cpp

7.29 prism::TX_prism_port Class Reference

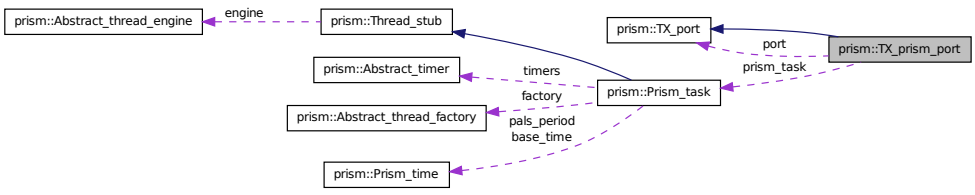
TX communication port in PRISM semantics.

`#include <prism_port.h>`

Inheritance diagram for prism::TX_prism_port:



Collaboration diagram for prism::TX_prism_port:



Public Member Functions

- TX_prism_port (TX_port *tx_port_arg, Prism_task *prism_task_arg, Delivery_property port_property, uint16_t target_timer_index_arg)

  Constructs a TX_prism_port object.

- virtual int32_t send (void *buf, size_t length) throw (Port_exception)

7.29.1 Detailed Description

TX communication port in PRISM semantics. TX_prism_port can be used as a general #TX_port, which has interfaces to send packets. However, it must be always coupled with #RX_prism_port on the other side. The logical time behavior of a packet sent by TX_prism_port is deterministic.

7.29.2 Constructor & Destructor Documentation

7.29.2.1 TX_prism_port::TX_prism_port (TX_port * *tx_port_arg*, Prism_task * *prism_task_arg*, Delivery_property *port_property*, uint16_t *target_timer_index_arg*)

Constructs a TX_prism_port object.

Notice that TX_prism_port only defines PRISM semantics not system specific network interfaces. Lower network layer defining system-specific network interfaces are defined by #tx_port_arg.

Parameters

tx_port_arg	TX port to send packets below PRISM layer. The object ownership is transferred to the TX_prism_port object; it is deleted when TX_prism_port object is deleted. Thereby, #tx_port_arg object must be created in heap.
prism_task_arg	#Prism_task using the port.

port_property	Defines target PALS round.
target_timer_index_arg	Defines the receiver timer index within the PALS round.

7.29.3 Member Function Documentation

7.29.3.1 int32_t TX_prism_port::send (void * *buf*, size_t *length*) throw (Port_exception) [virtual]

It attaches header defining when it must be delivered to the receiver. #TX_port::send()

Implements prism::TX_port.

The documentation for this class was generated from the following files:

- prism/include/prism_port.h
- prism/src/prism_port.cpp