RODOS - Introduction and Documentation

Sergio Montenegro, Anna Aumann

June 21, 2016

Abstract

This document has been created by compiling different documents and files belonging to the documentation of the RODOS real time operating system. It shall introduce the interested reader, experienced programmers as well as beginners, to the concepts of RODOS as operating system and enable them to use RODOS for their own projects. RODOS is open source and copyrighted by DLR and the university of Würzburg (2013). It already runs on several different micro controller and also on top of linux. RODOS is the perfect choice for projects, where unreliable subsystems are combined to a reliable complete system.

Chapter 1

Introduction

Abstract The first chapter introduces the RODOS real-time operation system. It explains the main concepts of RODOS and terms as active and passive objects, the middleware's publisher/subscriber principle as well as the programming interface. If you would like to have a quick start into RODOS, you are recommended to skip this chapter and go directly to chapter 2.

1.1 In General

RODOS is a real-time embedded operating system (OS) designed for applications demanding high dependability. Simplicity is its main strategy for achieving dependability, as complexity is the cause of most development faults. The system was developed in C++, using an object- oriented framework simple enough to be understood and applied in several application domains. Although targetting minimal complexity, no fundamental functionality is missing, as its microkernel provides support for resource management, thread synchronisation and communication, input/output and interrupts management. The system is fully preemptive and uses priority-based scheduling and round robin for same priority threads.

RODOS provides a middleware which carries out transparent communications between applications and computing nodes. The messages exchange is asynchronous, using the publisher-subscriber protocol. Using this approach, no fixed communication paths are established and the system can be reconfigured easily at run-time. For instance, several replicas of the same software can run in different nodes and publish the result using the same topic, without knowing each other. A voter may subscribe to that topic and vote on the correct result. The core of the middleware distributes messages only locally, but using the integrated gateways to a network based on the NetworkCentric protocol, messages can reach any node and application in the network. The communication in the whole system includes software applications, computing nodes and IO devices.

All communications in the system are based on the publisher/subscriber protocol: Publishers make messages public under a given topic. Subscribers (zero, one or more) to a given topic get all messages which are published under this topic. For this communication there is no difference in which node (computing unit or device) the publisher and subscribers are running. They may be in the same unit, or distributed around the network. They may be any combination of software tasks and hardware devices. To establish a transfer path, both the publisher and the subscriber must share the same topic. A Topic is a pair consisting of a data-type and an integer representing a topic identifier. Both the software middleware and network switch (called middleware switch), interpret the publisher/subscriber protocol in the same way.

The RODOS framework seeks to offer the simplest and smallest possible interface to user applications, while still providing all the required functionality and flexibility. It includes time management, CPU and memory management. The RODOS middleware provides communication between applications, networks and all devices attached to the network. The fault tolerance support implemented in the middleware allows us to create dependable systems using unreliable components. In its concept, a hardware failure is not an exception, but a normal case, which can be expected and has to be handled. RODOS redundancy management supports different strategies to provide the highest possible dependability, our target, which is the ultra high dependability using a principle which the world has since forgotten: Simplicity.

Simplicity does not mean, however, lack of functionality. Real time scheduling, resource management, synchronization, middleware and simple communication and all the functions one can expect from a microkernel are implemented – just as simply as possible. An important RODOS design target is the irreducible complexity; this is the minimal possible complexity for a determined function. When it becomes no longer possible to implement it simpler without destroying the functionality.

RODOS is based on very few and simple basic functions. Applications running on top of RODOS are implemented using object-oriented technology, resulting in highly modular application software. Applications running on the top of the RODOS middleware are built using the schema of software building blocks. Several (simple) building blocks (called applications) can be distributed and interconnected in a computer and devices network, to build more complex functionality. Building blocks can be implemented and tested independently of each other.

1.2 Core

RODOS was designed as a framework offering the following features:

1. object oriented C++ interfaces,
2. Ultra fast booting

3. real time priority controlled primitives multithreading,
4. time management (as a central point),
5. thread safe communication and synchronisation,
6. Event propagation

RODOS can be executed on embedded target hardware and on top of Linux. Applications can be moved from one host to another without modifications. The on-top-of-LINUX implementation helps developers to work locally on their workstation without having to use the target system. To move to the target, they have only to recompile the code. The behaviour is the same, except for timing requirements and time resolution, which on LINUX cannot be as exact as in the target systems.

Applications may be implemented by creating active and passive objects. Active objects may get CPU-time from the underlying core as reaction to time, to events, to message distribution and to requests from the object itself. To create an active object the user just needs to inherit and instantiate from the interface classes Thread, Event and/or instantiate Subscriber (Middleware Interface) objects.

The central element in the core is the time. The time begins at 0 (boot time) and increments continuously in nanosecond steps until "End-Of-Time" which is about 150 years into the future. This time controls almost all activities in the core.

Many threads may run (apparently) simultaneously. Each thread may run until it is suspended for a time period by itself (typically) or by another thread (not usual) (see figure 1.1). Also the access of a synchronised object may suspend its caller. Examples are entering a semaphore, reading from synchronised fifos, waiting for Messages, etc. .

A Thread may be suspended (blocked) until a time point, then it will be resumed automatically when the corresponding time point is reached. If a Thread shall be suspended without limit, we use the time point "End-Of-Time".

Any suspended thread may be resumed automatically at a time point or explicit at the moment. This may be done by other thread or by an event (time or interrupt server). Resuming a thread may be explicit or may be contained in a synchronised object. For example when leaving a semaphore, the leaving thread will see if another thread is waiting to enter. If so, the corresponding resume will be called. This is implemented inside of the semaphore, when leaving the semaphore. The core may activate event handlers too, to react to a time point or to an interrupt. Event handlers have to be very short, because they are executed in interrupt mode and while an event handler is being executed all other interrupts will be blocked. Event handlers may resume threads and manipulate data, like normal threads.

As said before many threads and event handlers may be executed apparently simultaneously. If we have only one CPU then only one Thread or one event-handler will be executed at any time point. Whenever an event handler shall be executed any possible

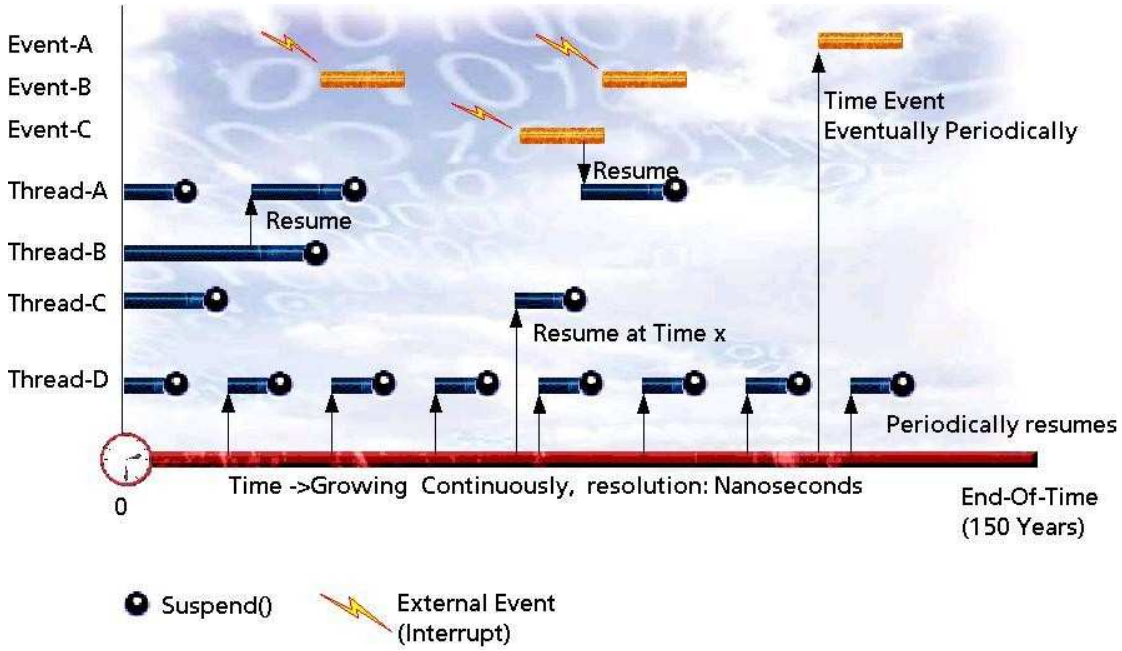


Figure 1.1: Suspending and resuming threads

running thread will be shortly interrupted to execute the handler and then the interrupted thread may continue. We can say events have the highest priority in the system. When an event handler is executed the hardware interrupts are blocked. The next event handler may be executed only after the current one returns (terminates); events handlers can not be interrupted.

From all threads which are ready to run (they are not suspended at the current time point) the one with the highest priority will be executed first. If another thread with higher priority becomes ready to run the current thread will be interrupted and the one with higher priority will be executed (preemption). If a thread is suspended, then the scheduler will search for the next one with the highest priority. At the end of the list is the idle thread which is always ready to run, but has priority 0. If two threads have the same priority, a round robin procedure will be used. In one sentence: a *fair priority controlled preemptive scheduling*. (see also figure 1.2).

1.3 Middleware

The RODOS middleware was designed to support fault tolerance. All threads (and applications) running on top of the RODOS middleware can exchange messages asynchronously using a publisher / subscriber protocol. The RODOS middleware distributes (and replicates) messages locally in each computing node and using gateways it may cross

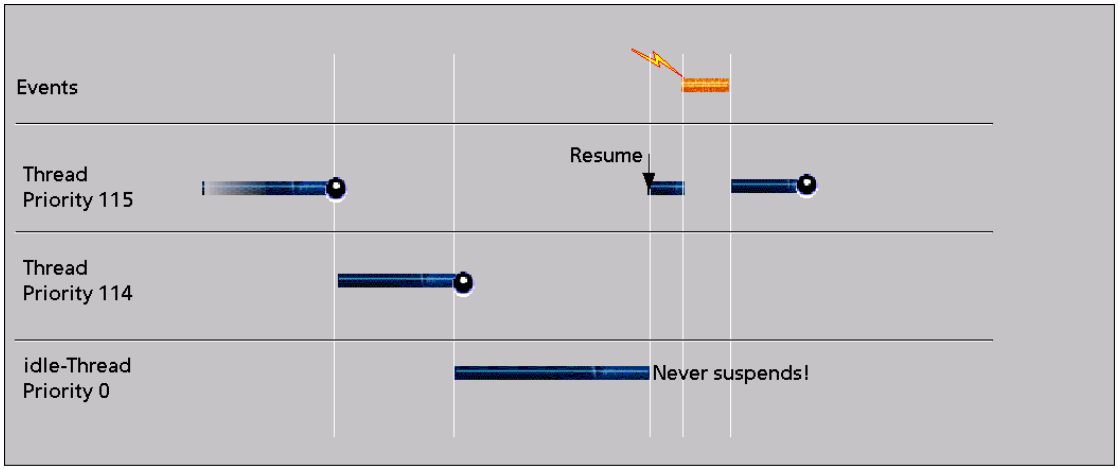


Figure 1.2: Preemptive CPU scheduling

node boundaries to reach all units in the network. Internally the middleware, gateways and hardware network (Middleware Switch) use all the same NetworkCentric protocol. Units attached to the network can be computing nodes and IO devices in the same way. The communication protocol is based on the most simple possible implementation of the publisher/subscriber protocol.

This provides very high flexibility and users do not have to differentiate between local/remote communication and between any combination of software/hardware/device communication. Communication relationships can be very dynamic. Units may disappear or appear, tasks may be migrated, activated or deactivated at any time. The position of applications can even change (migration) at runtime, without requiring any explicit reaction of the other involved applications. There are no fixed communication paths. Each data transfer is resolved just in time using the registered communication topics.

The middleware imposes no limitations on communication paths, but the user shall use/create a meaningful, reasonable and efficient inter-task communication structure. Publishers make messages public under a given topic. Subscribers (zero, one or more) to a given topic get all messages which are published under this topic. To establish a transfer path, both the publisher and subscriber have to share the same topic. A Topic is a pair of data-types and an integer representing a topic identifier. Both of the components - software middleware and network switch (called **middleware-switch**) interpret the same publisher/subscriber protocol in the same way (see figure 1.3).

Software applications may access an arbitrary number of topics, both as subscriber and as publisher. Devices, on the other hand, publish or subscribe typically only one topic and, according to the NetworkCentric model, are attached preferably to the hardware network. For very small systems or in exceptional circumstances devices may be attached to a computing node (but this is not recommended). In this case the device interface

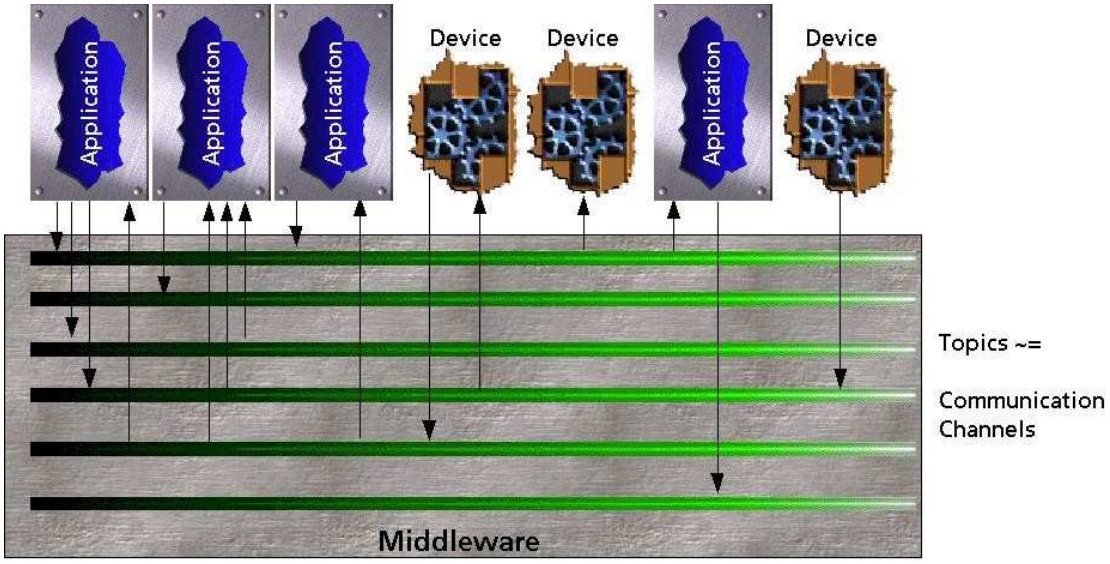


Figure 1.3: Topics in the Middleware

(IO) shall be encapsulated into a publisher/subscriber pair and attached to the local middleware. Complex functionality may be implemented as a network service which is implemented into applications and which may be distributed using topics. A topic may then be considered as a communication channel with an arbitrary number of writers and arbitrary number of readers and which may be distributed in different hardware units (see figure 1.4).

Threads and even event handlers may publish data in a non-blocking manner. As response to being published, the topic will search its list of subscribers and notify each of them. Each subscriber provides a putter object which handles the delivery of messages. It may just register the data or it may resume a waiting thread.

A local middleware distributes messages only locally. To access external units, computing nodes and devices one or more gateways have to be added to the local middleware. Gateways are smart publishers and subscribers which are not fixed to a single topic. They may listen to all topics and may publish messages in any topic. Gateways marshal internal middleware messages in an external format in order to send them to other hardware units. This external protocol may be, for example, a wireless protocol or Ethernet, SpaceWire, RS 322, MIL, radio communications, etc. The opposite, when the gateway gets messages from external units, it unmarshals the data and transforms it to the internal middleware format. Then using the corresponding topic it publishes the message locally. Using this model the network of services may be distributed in several computing units, several satellites and even between satellites and ground stations. This network would allow to substitute the typical command/telemetry procedures by a NetworkCentric network of services (see figure 1.5).

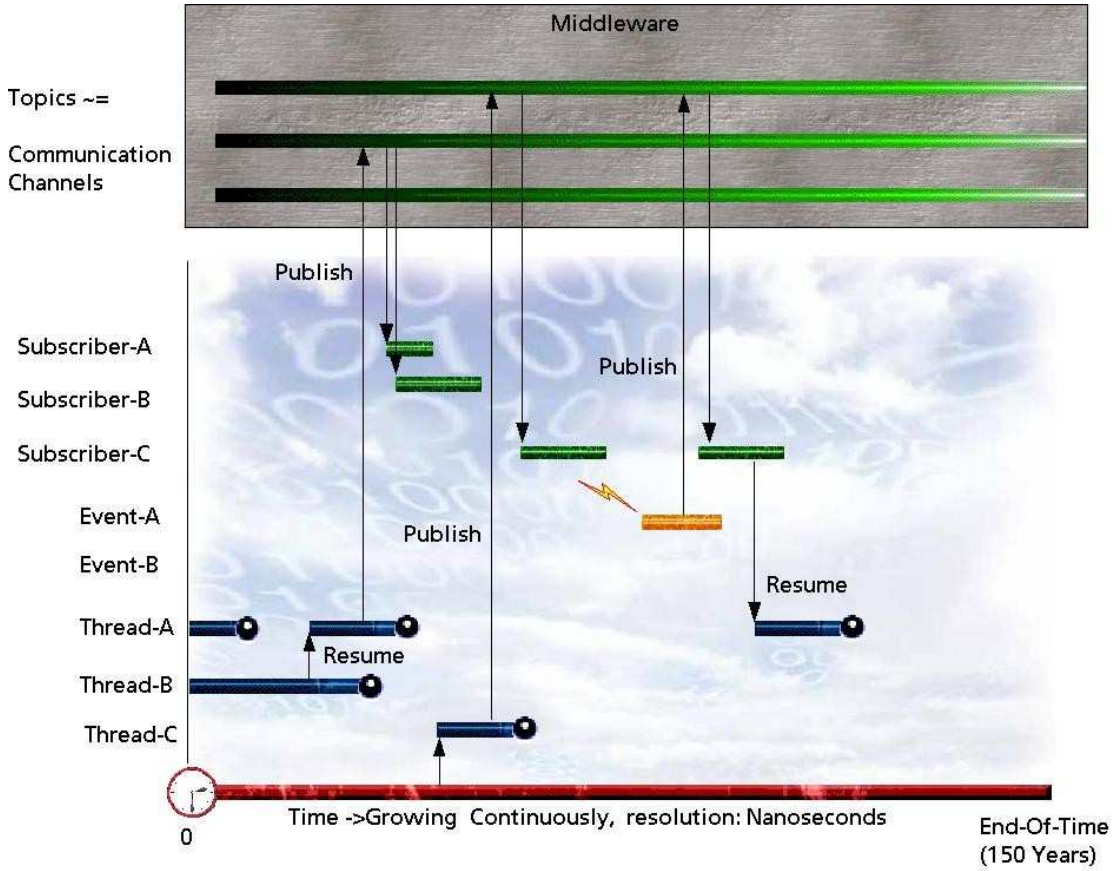


Figure 1.4: Message distribution using topics

The network is implemented using intelligent middleware switches, which can interface different protocols in order to be able to access different devices. A middleware switch implements internally the same topic protocol just like the software middleware. Each port of the middleware switch implements a (different) protocol converter, like a gateway, to access external units like for example computing nodes and IO devices. In the case of IO-Devices, the protocol converter substitutes the typical IO-Driver of conventional systems.

1.4 Applications and Building Blocks

To build a complex functionality and even perhaps many different functionalities in the same system, which is the typical case in satellites, it is advisable to encapsulate simple and clear cut related functions in building blocks – here called applications (see figure 1.6) – and to plug such blocks together thereby building a network of applications/building blocks (not to be confused with the hardware network). Adding devices to the network

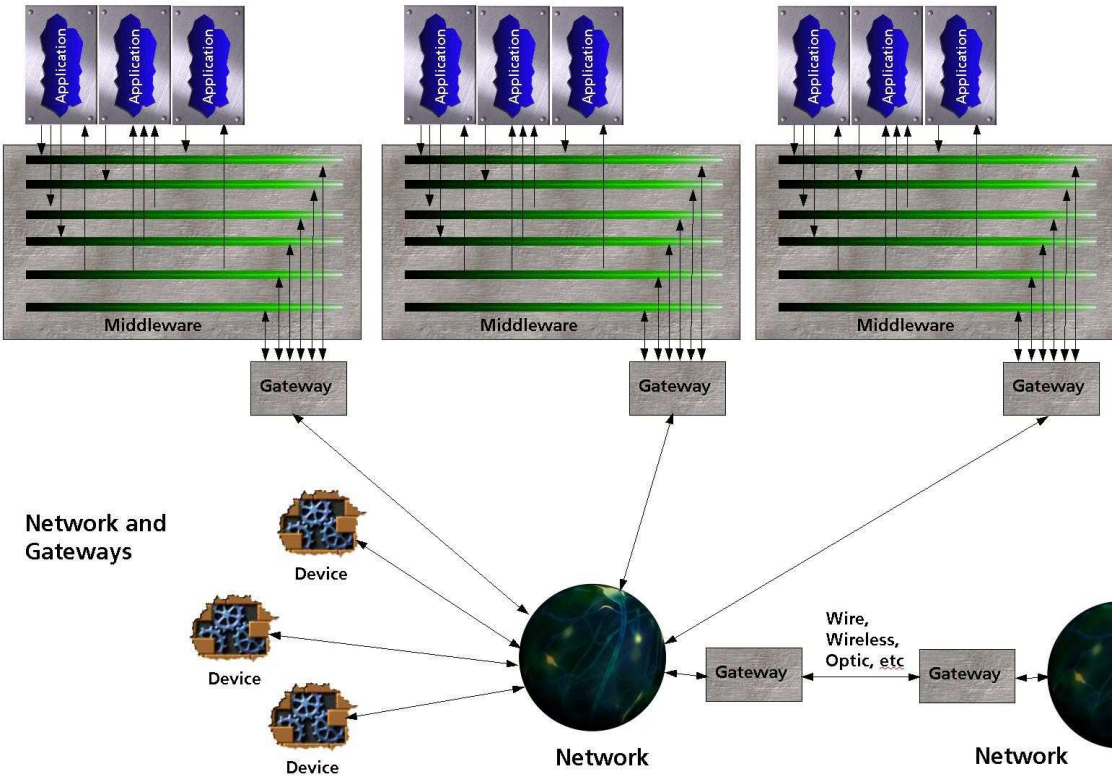


Figure 1.5: Gateways and networks

we get a network of services. Applications (or building blocks) may encapsulate threads, messages, event handlers, passive objects etc. From outside it shall not be visible to the innermost being of the application. The only interfaces to an application are messages which can be distributed/subscribed. Each applications shall provide one specific service to the system.

The network of services consist of applications and devices distributed in several computing units and hardware. Actuator devices do not produce information services but physical services. The opposite are the sensor devices, which build an interface from the physical world to the information world. Software applications stay (inputs and output) in the information world and the devices (sensors and actuators) are the bridge from the information world to the physical world.

There may be different system configurations for the distribution of applications in nodes and even all applications may run in the same node (typical configuration in small systems, see figure 1.7).

A services network may be extended (or modified) to provide more functionality without having to modify the individual building blocks. For example we may add instrumentation for debugging and a data logger by just adding new building blocks to the network

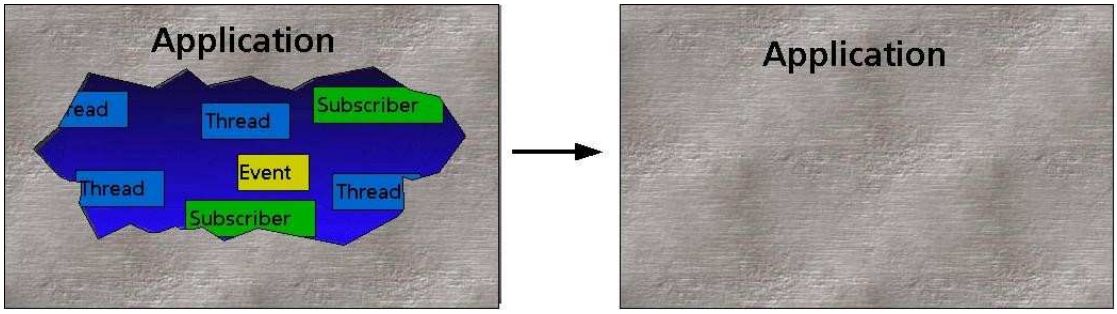


Figure 1.6: Applications encapsulate elemental components

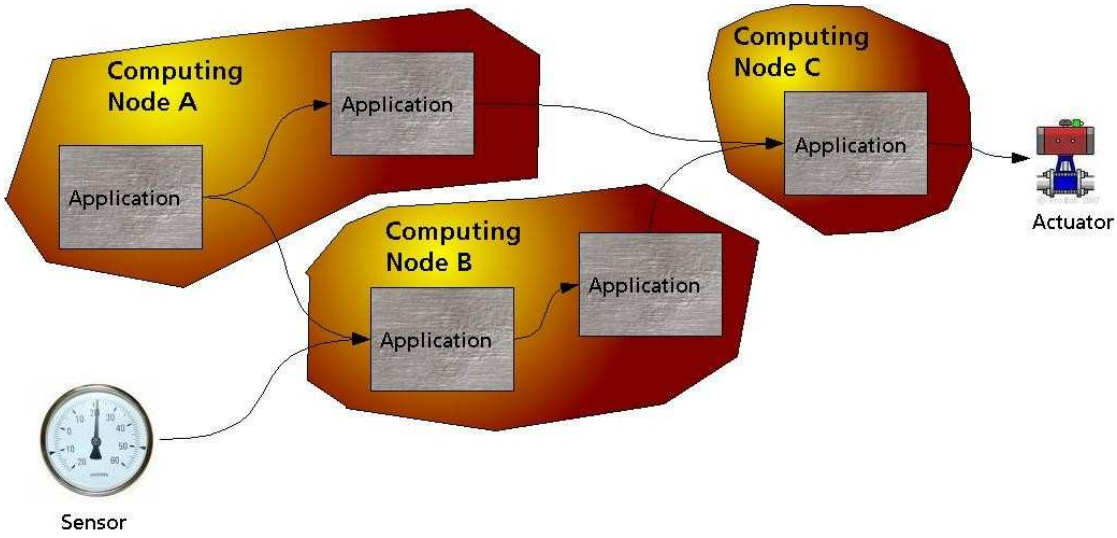


Figure 1.7: Example of distribution of applications

as shown in figure 1.8.

To build fault tolerance, both producers and consumers of services may be replicated. For the programmer there is no difference if one or more replicas are working in the same network. To merge outputs from several sources we need voters which may intercept messages, select the most probable correct one and forward it to the device. Even voters may be replicated (see figure 1.9).

1.5 Programming Interface

RODOS offers an integrated framework (OO) interface. Both core and middleware together will be called the RODOS-framework. The RODOS framework aims to offer the most simple and small as possible interface to user applications, which still provides

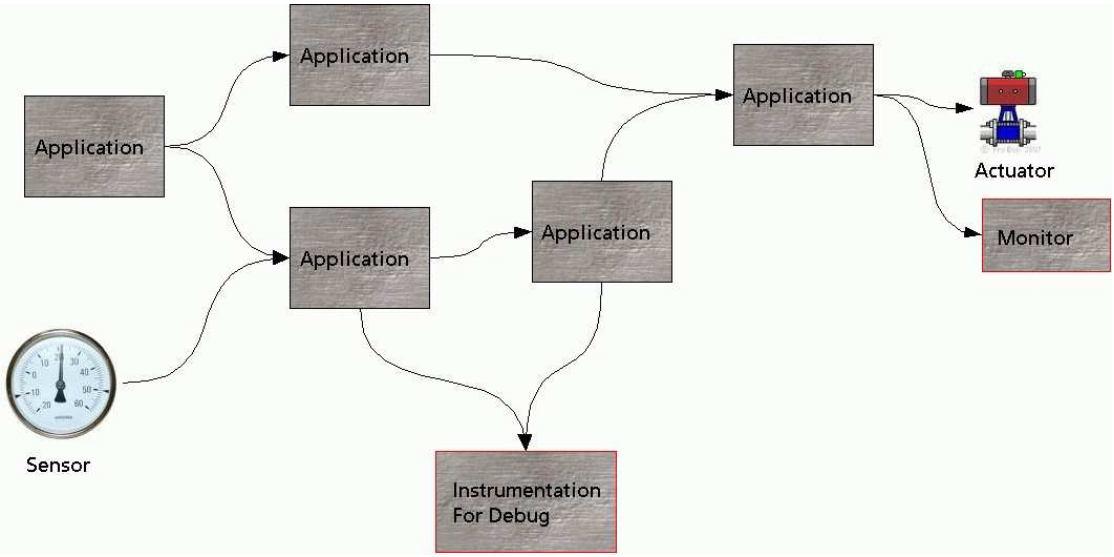


Figure 1.8: Adding applications to a network of services

all required functionality and flexibility.

The RODOS-framework includes time management, resource management and communication functionality. Without an application the framework is inactive. An application can add actions to the RODOS-framework by inheriting classes and creating active objects. These objects will be integrated automatically into the framework. In this way the framework will be extended with user functionality.

The framework technology is a further step following the object-oriented technology, where the functionality is provided by OO methods encapsulated in classes and other functionality by means of inheritance. A framework is composed of several classes in a structure with different relationships: inheritance, references and contention. The whole structure has a specific functionality. The user can adapt its functionality to his needs as follows: Some classes in the structure provide the adaptation interface (inheritance) for the user, while other classes offer just a function/method interface like the "normal" procedural interfaces. To adapt the functionality of the framework to his need, the user writes new classes, which inherit from the adaptation interface classes (subclasses). Some adaptation interface methods shall be overloaded with user methods and functionality in order to integrate the user functionality into the framework. The new (user) subclasses and its objects are integrated automatically (by inheritance) into the structure and the user does not need to register them manually. This makes the integration much simpler and reduces mistake sources.

A software system consists of a collection of passive and active objects. The passive objects just offers data and methods to be accessed by active objects. Some of these methods can, however, provide means for synchronisation for threads. Active objects

will be activated (executed) by the underlying system. They may get the CPU (time) as a reaction to time-points, events or requests. The most common example of active objects are the threads. Many threads may run (apparently) concurrently. Time is a very central point for the RODOS framework. Execution of threads is mainly controlled by time. Each thread can define periodic execution or just time points when it shall be executed. The RODOS framework will try to satisfy all these time requirements. If more than one thread requests the CPU for the same time, then the conflict will be resolved according to priorities. The thread with the higher priority will be executed next. There may be exclusive regions, where maximum one thread may execute a piece of code, or situations when a thread waits for a signal or data from another thread. RODOS provides means for such thread synchronisations, using semaphores, and synchronous data FIFOS. Another class of active objects are the events. The user can define zero or more eventhandlers to react to events. The most popular example of events are time events, which produce a reaction when a time point is reached. Another example is the reaction to external events (interrupts). User eventhandlers shall provide a handle function which will be called when the corresponding event arrives.

The last class of active objects are the middleware subscribers. A middleware subscriber may wait for a determined type of message (topic). When an expected message arrives, the corresponding subscriber will be activated.

The following picture is an example of user applications which extend threads and events to add functionality to the framework. The same can be done to subscribers. Passive objects like objects from the synchronisation classes may be instantiated and used without inheritance

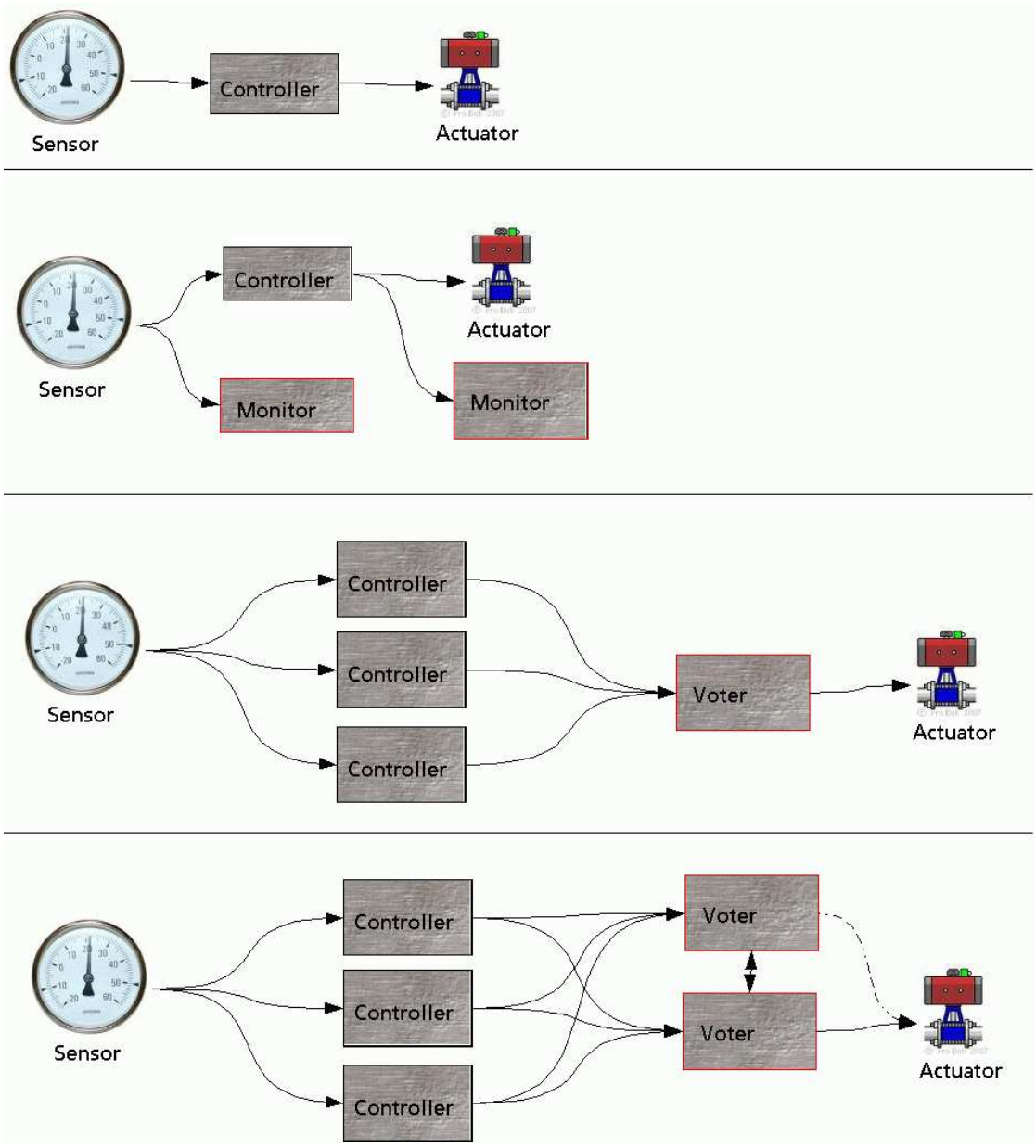


Figure 1.9: Adding fault tolerance by just plug and play

Chapter 2

The First Steps

Abstract Your RODOS distribution comes with many example programs. In the following chapter those programs found in the folder first-steps are presented in detail. After reading this chapter you should therefore be able to write and compile your first simple RODOS programs. If you like to have a more detailed insight you are then recommended to read the chapters afterwards. It is assumed that you are working on a Linux platform.

2.1 Before Start

The RODOS directory is structured into four folders: *make* contains build scripts which allow to compile RODOS applications for a variety of hardware platforms. The *api* folder contains all header files defining the application programming interface, you should refer to these files in order to see all possible RODOS functions (they are, however, partly, introduced in a later chapter). The *tutorials* folder contains the examples we want to try. There is an example for almost everything, so the first approach is usually to find an example before programming. More documentation of RODOS can be found in the *doc* folder, which should also contain a copy of this documentation. Last, the *src* folder contains the RODOS core source files which are not important for a RODOS user.

A RODOS program or application usually consists of more than one source file. In order to compile an application some steps have to be executed first.

1. Open a Terminal
2. Enter the RODOS root directory
3. Set some shell variables that are needed by the compile scripts
   $ source make/rodosenvs
   It has to be executed every time when opening a new terminal!

4. Compile the RODOS library for a Linux x86 PC
   $ linux-lib
   Has to be done only once for every RODOS version, unless something in folder src or api has been modified.
5. Enter the folder with the user program
   $ cd tutorials/first-steps
6. Compile the user program
   $ linux-executable usercode1.cpp usercode2.cpp...
7. Execute the binary
   $ ./tst
8. Exit the program with Ctrl+C

As a shortcut, a file has been created for every example that compiles the necessary code-files and executes it (e.g. *execute-example-01* for the example in chapter 1). Attention: Don't forget to do step 1 to 5 beforehand.

2.2 Hello World

Listing 2.1: HelloWorld.cpp

```
1 #include "rodos.h"

3 class HelloWorld : public Thread {
4   void run(){
5     PRINTF("Hello World!\n");
6   }
7 } helloworld;
```

The *HelloWorld* class is our first example of a RODOS threads. A thread is a small building block of a RODOS application. Threads execute tasks once or repeatedly or react on input or events. The `HelloWorld` class is derived from the `Thread` class. In order to use the RODOS functions and classes, the RODOS header `rodos.h` has to be included on top of the file (line 1). Every thread has to implement the virtual `run()` function which contains the actual thread task (line 4). The `HelloWorld` class only prints the string `"Hello World"` once (line 5). Threads are not instantiated at runtime, but as global variables. At compilation time they are automatically added to the RODOS scheduler and called according to their task. The example *helloworld-multiple* is a good illustration of how a scheduler can suspend a single thread in favour of another.

2.3 Basic

Listing 2.2: basic.cpp

```
1  #include "rodos.h"

3  static Application appHW("HelloWorld");

5  class HelloWorld : public Thread {

7  public:

9    HelloWorld() : Thread("HelloWorld") { }

11   void init() {
12     PRINTF("Printing Hello World");
13   }

15   void run(){
16     PRINTF("Hello World!\n");
17   }
18 };

20 HelloWorld helloworld;
```

The example *basic.cpp* extends the former HelloWorld class and implements the basic structure of a RODOS programm. Threads can be grouped under a common application. Here the application is called `appHW` and named as `"HelloWorld"` (line 3). A constructor has been added to the `HelloWorld` class which gives the thread a name (line 9). Another constructor parameter is the thread priority. Threads with higher priorities get assigned the computing resources when more than one thread is runnable at a time. Besides the `init` function has been implemented (line 11). The method is called after all constructors are executed and before the threads are activated. It should be overloaded in case the thread has to call other objects for the initialization, hence after all instances are present and before any run method is invoked.

2.4 Time

Listing 2.3: time.cpp

```
1  #include "rodos.h"

3  static Application module02("TestTimeAT");

5  static class TestTime : public Thread {

7  public:

9    TestTime() : Thread("waitAT") { }

11   void run(){
```

```
13      PRINTF("waiting until 3rd second after start\n");
14      AT(3*SECONDS);
15      PRINTF("after 3rd second\n");

17      PRINTF("waiting until 1 second has pased\n");
18      AT(NOW()+1*SECONDS);
19      PRINTF("1 second has pased\n");

21      PRINTF("print every 2 seconds, start at 5 seconds\n");
22      TIME_LOOP(5*SECONDS, 2*SECONDS){
23        PRINTF("current time: %3.9f\n", SECONDS_NOW());
24      }

26    }

28    void init() { PRINTF("Waiting time"); }

30  } testTime;
```

This example shows how time dependent processes can be modeled in RODOS. It demonstrates how to do something at a specific point in time, after a defined amount of time and periodically. While the thread waits for the defined time, other threads can be executed. Time in RODOS is defined with a long long type `TTime` and represents the number of nanoseconds elapsed since startup. RODOS offers helpful macros to simplify the timing of threads. `NOW()` returns the current time in nanoseconds, whereas `SECONDS_NOW` ↪ `()` returns the current time in seconds. To suspend the calling thread until a given point in time is reached, `AT(time)` is used where `time` is an absolute point (line 14). In combination with `NOW()` or `SECONDS_NOW()` a relative point can be simulated (line 18). If a task needs to be executed periodically, `TIME_LOOP(firstExecution,Period)` provides a loop with no end (line 22).

The units of time are also available as macros (`NANOSECONDS`, `MICROSECONDS`, `MILLISECONDS`, `SECONDS`, `MINUTES`, `HOURS` and `DAYS`, `WEEKS`).The highest time possible is `END_OF_TIME` (which is used for example to suspend a thread for a time not specified exactly).

2.5 Priority

Listing 2.4: priority.cpp

```
1  #include "rodos.h"

3  static Application module01("Priority");

5  class HighPriorityThread: public Thread{
6  public:
7    HighPriorityThread() : Thread("HiPriority", 25) { }

9    void init() {
```

```
10      PRINTF(" hipri = '*'");
11    }

13    void run() {
14      while(1) {
15        PRINTF("*");
16        AT(NOW() + 1*SECONDS);
17      }
18    }
19  };


22  class LowPriorityThread: public Thread {
23  public:
24    LowPriorityThread() : Thread("LowPriority", 10) { }

26    void init() {
27      PRINTF(" lopri = '.'");
28    }

30    void run() {
31      long long cnt = 0;
32      while(1) {
33        cnt++;
34        if (cnt % 10000000 == 0) {
35          PRINTF(".");
36        }
37      }
38    }
39  };

41  HighPriorityThread highPriorityThread;
42  LowPriorityThread  lowPriorityThread;
```

In RODOS it is possible to define threads with higher and threads with lower priorities. If the thread with highest priority runs, the other threads will wait. In RODOS is a higher priority defined with a higher number. The lowest priority is 1, the highest is 2^{31}. The priority of a thread is defined in the thread constructor.

In this example two threads, one with a high priority which is executed very shortly every second and one with a low priority which is executed constantly. The high priority thread (printing "*" in line 15) runs when one second is over, although the low priority thread (printing "." in line 35) does not suspend. The output then looks as following.

```
...
Threads in System:
Prio =
0 Stack = 32000 IdleThread: yields all the time
Prio =
10 Stack = 32000 LowPriority: lopri = '.'
Prio =
25 Stack = 32000 HiPriority: hipri = '*'
```

```
BigEndianity = 0, cpu-Arc = x86, Basis-Os = baremetal, Cpu-Speed
(K-Loops/sec) = 350000
-----------------------------------------------------
Default internal MAIN
--------------- application running ------------
*.........*.........*.........*..........*.........*....
```

If a thread needs to do something without being interrupted priority ceiling is possible by wrapping some code with the `PRIORITY_CEILING` command. The wrapped code is executed in highest priority possible as demonstrated in the file priority_ceiler.cpp. At first it is the same as the priority.cpp example but after leaving the first while-loop priority ceiling is activated. The following code will never be interrupted by the high prioritiy thread. Compile the tutorial priority_ceiling.cpp in `tutorials/first-steps` as described in section 2.1 and execute it. The output should be the following:

```
*.....*....*.....................................
```

2.6 Thread Communication

So far every thread had a task on its own. Now we want them to communicate with each other. As threads are working concurrently not using a dedicated data structure which is thread safe is dangerous and may lead to inconsistent data, because the data might be half written in the shared variable when the thread is interrupted. RODOS offers three different types of data structures, each useful for its own use case. A `CommBuffer` is a double buffer with only one writer and only one reader. Both can work concurrently. The writer may write at any time. The reader gets the newest consistent data (eg. the last complete written record). The type of the CommBuffer can be defined arbitrarily.

Listing 2.5: commbuffer.cpp

```
1  #include "rodos.h"
2
3  static Application applic("ComBufTest");
4
5  CommBuffer<int> buf;
6
7  class Sender : public Thread {
8     void run () {
9      int  cnt = 0;
10     while(1) {
11       cnt++;
12       PRINTF("Writing %d\n", cnt);
13       buf.put(cnt);
14       AT(NOW() + 3*SECONDS);
15     }
16   }
17 };
18
19 class Receiver : public Thread {
20   void run () {
```

```
21      int cnt;
22      while(1) {
23        buf.get(cnt);
24        PRINTF("Reading %d\n", cnt);
25        AT(NOW() + 2*SECONDS);
26      }
27    }
28  };

30  Sender   sender;
31  Receiver receiver;
```

Example 2.5 shows the use of a commbuffer which stores the type `int`. It has to be outside of the threads so both, the reader as well as the writer, can access it (line 5). The Sender thread puts its counter value every three seconds (line 13), while the Receiver threads reads every two seconds (line 25). Changing this to a value greater than three seconds shows, that only the most recent value is read using a CommBuffer. If every value is needed by the receiver, a FIFO has to be used (based on the principle of a queue). Writing to a full fifo has no effect and returns 0. Reading from an empty fifo returns 0. The first value inserted into the fifo will be the first value to be read.

Listing 2.6: fifo.cpp

```
1   #include "rodos.h"

3   static Application applic("FifoTest");

5   Fifo<int, 10> fifo;

7   class Sender : public Thread {
8     void run () {
9       int  cnt = 0;
10      PRINTF("sender\n");
11      while(1) {
12        cnt++;
13        bool ok = fifo.put(cnt);
14        if (ok) {
15          PRINTF("Sending %d\n", cnt);
16        } else {
17          PRINTF("Fifo full\n");
18        }
19        if ((cnt % 15) == 0) {
20          PRINTF("Wainting 3 seconds\n");
21    AT(NOW() + 3*SECONDS);
22        }
23      }
24    }
25  };

27  class Receiver : public Thread {
28    void run () {
29      int cnt;
30      PRINTF("receiver\n");
```

```
32      while(1) {
33        bool ok = fifo.get(cnt);
34        if (ok) {
35          PRINTF("reading %d\n", cnt);
36        } else {
37          AT(NOW() + 1*SECONDS);
38        }
39      }
40    }
41  };


44  Sender   sender;
45  Receiver receiver;
```

If the data has to be processed short times after sending it, a SyncFifo (the third data structure for thread communication) is a good option. In this case the sender will be suspended if the fifo is full and the receiver will be suspended until data is ready. (see example fifo_sync.cpp).

2.7 Critical sections

To avoid concurrent access of critical sections semaphores have to be used. A semaphore is defined outside of threads. There are two options to protect a critical section using a semaphore. Either one can use the semaphore functions *enter()* and *leave()* (see example semaphore.cpp) or use the macro `PROTECT_WITH_SEMAPHORE(sema)` (see example semaphore_macro.cpp). However, a deadlock may occur when using a semaphore (see example semaphore_deadlock.cpp).

2.8 Events

Events can be used to react to interrupts from timers and signals from devices. Do not use them for complex actions, because they cannot be interrupted. However they are used to trigger threads that handle the interrupts (by calling `thread.resume()`). Implement them as short as possible. An event has basically two methods: The `init()` method simillar to threads and the `handle()` method in which the code is defined that handles the event. Events are activated periodically using `activatePeriodic(startTime, period)` or at a given point in time using `activateAt(time)`.

Listing 2.7: event.cpp

```
1  #include "rodos.h"

3  static Application module01("resumefromEvent");

5  class TestWaiter: public Thread {
6  public:
7    void run(){
8      while(1){
```

```
9         PRINTF("Suspend and wait until some one resumes me\n");
10        AT();
11        PRINTF("testwaiter running again at %3.9f\n", SECONDS_NOW());
12      }
13    }
14  };


17  static TestWaiter testWaiter;


20  class TimeEventTest : public TimeEvent {
21  public:
22    void handle(){
23      xprintf("   Time Event at %3.9f\n", SECONDS_NOW());
24      testWaiter.resume();
25      xprintf("   Testwaiter resumed from me\n");
26    }

28    void init() { activatePeriodic(5*SECONDS, 3*SECONDS); }
29  };

31  static TimeEventTest te01;
```

The example *event.cpp* contains a thread testWaiter suspends (line 10) until it gets resumed by the time event te01 (line 24). Please note how `AT()` is used in line 10 instead of `suspendCallerUntil (END_OF_TIME)`. The activation setting for the time event is set by overriding the `init()` function (line 28). The output is according to the expectations. Testwaiter prints its first line and then suspends. The time event, (first at 5 seconds), then resumes the waiting thread.

```
Suspend and wait until some one resumes me
   Time Event at 5.000107974
   Testwaiter resumed from me
testwaiter running again at   5.000135475
Suspend and wait until some one resumes me
   Time Event at 8.000168306
       Testwaiter resumed from me
```

2.9 Middleware

So far threads either only used local data and variables or data exchange happened between one writer and one reader using CommBuffer or Fifo. For communication between different tasks or even between tasks of different RODOS nodes the middleware is used. The RODOS middleware is based on a publisher/subscriber protocol. This allows programmers to distribute data without notion of the other side. Using a network interface and the corresponding gateways, the middleware may make the node borders transparent. Applications running on different computers may communicate as if they were on the same computer. Any thread can publish messages under a given topic, while subscribers of the same topic receive the published data. There is no direct connection from a sender to a receiver. All subscribers are registered in a list. Each time a message is published the list of all subscribers will be searched and for each

subscriber where the topic matches the associated putter will be called to store a copy of the published message. A topic is a pair of a data-type and the topic id, if the topic id is "-1" the id will be generated. For example, the topic `counter1` is used to share data of type long.

```
Topic<long> counter1(10, "counter1");
```

The following code example this time consists of more than one source file. The topics used are defined in *topics.cpp*, the publisher or sender thread is defined in *sender.cpp*. There are three different subscriber examples. The first thread inherits from the `Subscriber` class and has to implement the `put(...)` function. The second thread uses a CommBuffer to store the received topic. The third thread inherits from `Putter`.

If the receiver needs only the latest data and has to be executed periodically, the CommBuffer solution should be used. For synchronized communication the subscriber put method in combination with resuming a thread is the way to do it, a SyncFifo is a good alternative. To receive from multiple topics with one method a Putter should be used.

Listing 2.8: sender.cpp

```
1  #include "rodos.h"
2  #include "topics.h"

4  static Application senderName("Publisher 01 simple", 1100);

6  class MyPublisher01 : public Thread {
7  public:
8    MyPublisher01() : Thread("SenderSimple") { }

10   void run () {
11     long cnt = 0;
12     TIME_LOOP(3*SECONDS, 3*SECONDS) {
13       PRINTF("Publisher01 sending Counter1 %ld\n", ++cnt);
14       counter1.publish(cnt);
15     }
16   }
17 } publisher01;
```

The publisher thread in the *sender.cpp* example shows how new data is send over a topic. The topic definition has to be included (line 2). The incremented counter is then published every three seconds (line 14) by a call to `counter1.publish(cnt);`

Listing 2.9: receiver_simple.cpp

```
1  #include "rodos.h"
2  #include "topics.h"

4  static Application  reciverSimple("ReciverSimple", 1100);

6  class SimpleSub :  public Subscriber {
7  public:
8    SimpleSub() : Subscriber(counter1, "simplesub") { }
9    long put(const long topicId, const long len, const void* data, const
       ↪ NetMsgInfo& netMsgInfo) {
```

```
10      PRINTF("SimpleSub - Length: %ld Data: %ld TopicId: %ld \n", len, *(
            ↪ long*)data, topicId);
11      return 1;
12    }
13  } simpleSub; // try this: , a, b, c;
```

This is the first receiver example which is derived by the `Subscriber` class. The `SimpleSub` subscriber is initialized with the topic to subscribe to (line 8). In the Commbuffer subscriber (file *receiver_commbuffer.cpp*) an instance of Subscriber is created using a CommBuffer directly as seen in listing 2.10.

Listing 2.10: receiver_commbuffer.cpp

```
1    static CommBuffer<long> buf;
2    static Subscriber receiverBuf(counter1, buf, "receiverbuf");
```

Further examples are found in the folders *alice_bob_charly* and the various *middleware* folders. A more advanced example is found in the last chapter of this documentation (chap. 5).

Chapter 3

The RODOS API

Abstract Whereas the previous chapter focused on giving you a good start into programming RODOS applications, this chapter more detailed presents the RODOS application programming interface. The API files are in the *api* directory of your RODOS core folder. The following chapter can therefore be used as a reference when looking for certain RODOS functions

3.1 Core

Time Model

As a real time operating system, RODOS relies on a dependent model of time. The API offers the `TimeModel` class to define a custom model, however the user is advised to use the global object `sysTime`.

Time in RODOS is stored as a `int64_t` value. Several time constants are implemented (see Listing 3.1). They are used when suspending a thread or defining a thread period. The constant `END_OF_TIME` refers to a point in time approx. 150 years from now.

Listing 3.1: Time Constants

```
1        #define END_OF_TIME    0x7FFFFFFFFFFFFFFFLL

3        #define NANOSECONDS     1LL
4        #define MICROSECONDS    (1000LL * NANOSECONDS)
5        #define MILLISECONDS    (1000LL * MICROSECONDS)
6        #define SECONDS         (1000LL * MILLISECONDS)
7        #define MINUTES         (60LL * SECONDS)
8        #define HOURS           (60LL * MINUTES)
9        #define DAYS            (24LL * HOURS)
10       #define WEEKS           (7LL  * DAYS)
```

The clock model class holds the parameters of a linear clock model that relates UTC to the Primary OnBoard Time (t). UTC = utcDeltaTime + localTime + drift*(localTime - tSync)

Here utcDeltaTime = UTC - localTime is determined from thw difference of the UTC time and the Primary OnBoard Time at an instance t = tSync. In the absence of a drift error, utcDeltaTime matches the actual value of UTC at the start of the onboard clock, i.e. at t=0. The drift value (dUTC/dt)-1 measures the fractional rate error of the onboard clock.

There are several conversion methods between different time standards. For example:

- between local time (system time) and Universal Time Coordinated (UTC)
```
int64_t localTime2UTC(const int64_t loaclTime)
int64_t UTC2LocalTime(const int64_t utc)
```
- between local time (time units since 1. January 2000) and calendar [1]
```
static int64_t calendar2LocalTime(int32_t year, int32_t month, int32_t day, int32_t
↪  hour, int32_t min, double sec)
void localTime2Calendar(const int64_t localTime, int32_t& year, int32_t& month, int32_t
↪ & day, int32_t& hour, int32_t& min, double& sec)
```
- between local time and modified julian date
```
static double localTime2mjd_UTC(const int64_t &localTime)
```
- between calender and modified julian date
```
static double calendar2mjd_UTC( const uint16_t &year, const uint8_t &month, const uint8_t
↪  &day, const uint8_t &hour, const uint8_t &minute, const double &second )
static void mjd_UTC2calendar( const double &MJD_UTC, uint16_t &year, uint8_t &month
↪ , uint8_t &day, uint8_t &hour, uint8_t &minute, double &second )
```
- between modified julian date in UTC format to UT1 or TT format
```
static double mjd_UT1fromUTC( const double &MJD_UTC, const double &UT1_UTC )
static double mjd_TTfromUT1( const double &MJD_UT1, const double &UT1_UTC, const double
↪  &UTC_TAI )
```

As the current time is often used, there are two macros to refer to the nanoseconds or seconds since system start.

```
#define NOW()TimeModel::getNanoseconds()
#define SECONDS_NOW()((double)TimeModel::getNanoseconds()/ (double)SECONDS)
```

ListElement

RODOS organizes static elements as threads, applications and events as single linked lists using its own implementation of `ListElement`. This class is not thread safe and should only be used for static elements which will never be deleted. The list itself is a pointer to a `ListElement`.

`ListElements` offers only a few methods to append an element to a existing list, as well as getting its name and the next element in the list (see listing 3.2).

[1] Reference : van Flandern T.C., Pulkinnen K.F.; Low-Precision Formulae for Planetary Positions; Astrophys. Journ. Suppl. vol. 41, pp. 391 (1979)

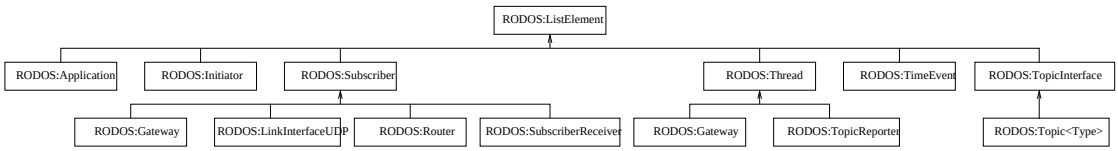


Figure 3.1: Classes inheriting from ListElement

Listing 3.2: Methods of ListElement

```
1    ListElement(List &list, const char* name = "ListElement", void*
         ↪ myOwner = 0);

3    void append(List& list);

5    const char* getName()
6    ListElement* getNext()
```

The classes inheriting from `ListElement` are displayed in figure 3.1. Applications, threads, time events and initiators are explained in this section below.

Application

Complex user tasks sometimes are split up into more than one thread. For example, the read-out and pre-processing of data might need several threads: one to poll the data from the sensor, a thread which processes the raw data and another one to forward the sensor data. Several threads and other active objects can be grouped together as an application which is defined through a name and an ID. Other applications can use this to find out, if the application has been linked/loaded or not. The IDs from 1000 up to 2^{31} are reserved as user IDs. RODOS intern topics are assigned IDs from 0 to 99. Input/output applications (e.g. UDP) are assigned Ids between 100 and 999.

```
Application(const char* name, const int32_t id=1000)
```

In order to find an application by its ID the static method `findApplication` is used. In case no application with the given ID is found, the method returns `code` null. The function shall not be called before the main function.

```
static Application* findApplication(const int32_t id)
```

The application class offers three different virtual functions to be overwritten by the user. It is possible to overwrite functions in order to activate or deactivate the application, which could for example suspend all threads belonging to the application.

Listing 3.3: Virtual Functions of an application

```
1      virtual void init(void) { } ///< called by rodos::main at system
           ↪ start
```

```
3       virtual void activate(); ///< user implemented to activate the
            ↪ application
4       virtual void deactivate(); ///< user implemented to deactivate the
            ↪ application
```

The `request` function is implemented to be able to send requests to the application. In case it is not implemented it returns `-1`, in case of an error `0`.

```
virtual int32_t request(void* requestMsg, int len, void* answerMsg, int maxLen)
```

Thread

A Thread is the base class for a user process, which will be executed according to a priority assigned. It provides stack and context management, and can be recognized by its name.

A Thread is an active schedulable object with own context and stack. Most RODOS applications consist of at least one object of a class derived from Thread. Such a class has to implement the run method, which is the entry point for user code. The Thread class itself is derived from ListElement as all threads are registered in a list at the start of the application. Therefore threads cannot be generated dynamically.

```
Thread(const char* name = "AnonymThread", const long priority = DEFAULT_THREAD_PRIORITY,
↪ const long stackSize = DEFAULT_STACKSIZE)
```

A thread is characterized by its name, priority and stack size. Only the thread with the highest value from several runnable threads at a time get computing resources. The size of the stack memory for the thread is defined through `stackSize`. Choose a matching value for the task of the thread.

```
virtual void run()
```

```
virtual void init()
```

In case of other objects to be called for initialization by the thread, besides of `run()` also the `init()` method should be implemented. The method is called after all constructors are executed and before the threads are activated by a call to the method run.

Usually a thread executes its task only at specific times or when an event occured (for example incoming data). At other times it can be suspended. There exists two functions in order to suspend the calling thread.

```
static bool suspendCallerUntil(const TTime reactivationTime = END_OF_TIME, void* signaler
↪  = 0)
```

This functions suspends a thread until the given point of time or it is resumed by another thread or event. If a thread has to execute its task periodically instead *suspendUntilNextBeat* is used. The time of the first execution and the period has to be defined first.

Listing 3.4: periodicBeat

```
1    void suspendUntilNextBeat()
```

```
3    void setPeriodicBeat(const TTime begin, const TTime period)
```

For both suspending variants there exists shortcuts.

Listing 3.5: Shortcuts

```
1    #define TIME_LOOP(_begin, _period) for(setPeriodicBeat(_begin,
         ↪ _period);suspendUntilNextBeat(), true; )

3    #define AT(_time) { Thread::suspendCallerUntil(_time); }
```

TimeEvent

Instead of waiting on a fixed point in time threads are also able to wait on a `TimeEvent`. Such an event can be activated once at a specified point in time or periodically. A thread will wait on an event if it calls `suspendUntil(...)` with the parameter signaler as a pointer to that event. As threads and applications, TimeEvents are stored as lists and their activation handled by the RODOS core.

Semaphore

A semaphore is used to control the access of multiple processes (for example threads) to a common variable. It prevents that atomic tasks are interrupted and obsolete data is read. Next to its constructor, the two main functions of a semaphore are `enter()` and `leave()`. A caller will be blocked if the semaphore is occupied, however the owner may reenter the semaphore without deadlock. Leaving will not block the caller, but resume a waiting thread. A macro has been defined to ease the use of semaphores.

Listing 3.6: Semaphore

```
1    private:
2      Thread* owner;
3      long ownerPriority;
4      void* context; ///< used only on posix and on host-os
5      int ownerEnterCnt;

7    public:
8      Semaphore();

10     void enter();

12     void leave();

14     inline bool enterRetTrue() { enter(); return true; }
15   };

17   #define PROTECT_WITH_SEMAPHORE(sem) \
18     for(bool _onlyonce_ = sem.enterRetTrue(); _onlyonce_ ; _onlyonce_ =
         ↪  false, sem.leave())
```

3.2 Middleware

Topic

A topic is a communication channel. Middleware communication is based on the Publisher/Subscriber protocol. Publishers make messages under a given topic. Subscribers get all published messages. A topic consists of a data type and an integer representing the topic identifier. This topic id can be generated automatically according to the name used in the constructor, if -1 is given as a parameter.

```
Topic::Topic(long id, const char* name): TopicInterface(id, sizeof(Type), name)
```

A topic is created with a id and a name. The name is used for debug purposes and as a hash value, in case id has been set to -1.

```
unsigned long Topic::publish(Type &msg, bool shallSendToNetwork = true)
```

In order to distribute messages a publisher has to call the `publish(...)` function. The pointer to `msg` then will be distributed and its content might be modified by any subscriber. If `shallSendToNetwork` ↪ is set, then the message will also be forwarded to gateways which enables distributed communication.

```
unsigned long Topic::publishConst(Type msgConst, bool shallSendToNetwork = true)
```

To publish constant values, the method `publishConst(...)` is called (only basic data types as char, short, long, float double).

```
unsigned long Topic::publishMsgPart(Type &msg, unsigned int lenToSend, bool shallSendToNetwork
↪  = true)
```

If only a part of a message is to be sent, specified by `lenToSend`, this method is called.

The counterpart of publishing a message, is receiving a message as a subscriber.

```
unsigned long Topic::requestLocal(Type &msg)
```

Instead of sending data, the caller of this function expects data however the internal mechanisms are identical.The middleware will call putters from subscribers, which shall provide data instead of getting it.

Often a generic type for topics is needed. RODOS offers the `GenericMsgRef` struct which contains only a pointer and a length.

Listing 3.7: GenericMsgRef

```
1       struct GenericMsgRef {
2           long   context;
3           char* msgPtr;
4           int    msgLen;

6           GenericMsgRef() { msgLen = context = 0; msgPtr = 0; }
7       };
```

Subscriber

A subscriber is a link between a Topic and the receiver of a message, which can be a subclass derived from Putter or Thread. A subscriber is bound to one topic only, which is established during program start. Each subscriber has a reference to the associated topic and a putter to store messages. All subscribers are registered in a list. Each time a messages is published the list of all subscriber will be search and for each subscriber where the topic matches the associated putter will be called to store a copy of the published message.

There are two different possibilities to construct a subscriber:

```
Subscriber::Subscriber(TopicInterface &topic, Putter& receiver, const char* name = "anonymSubscriber
↪ ", bool isAGatewayParam=false)
```

This constructor is more complex. It is given the topic to which the owner will be subscribed, as well as a reference to the receiver of the topic messages. A subscriber can also be given a name. Gateways get from all topics and they decide wheter to forward or not.

```
Subscriber::Subscriber(TopicInterface &topic, const char* name = "anonymThreadSubscriber"
↪ )
```

A more customized subscriber is created by only supplying the subscribed topic and, possibly, a name. This subscriber has to redefine the `virtual long Subscriber::put(const long topicId, ↪ const long len, const void* data, const NetMsgInfo& netMsgInfo)` method.

Putter

Putters are receivers of a message or used for data transfer and buffering in general. Its a virtual super class which offers only one method to override:

```
 virtual bool Putter::putGeneric(const long topicId, const unsigned int len, const void*
↪ msg, const NetMsgInfo& netMsgInfo)
```

It is used to store data, in order to have a generic interface to data storage. RODOS has implemented several kinds of FIFOs (First In First Out data structures) as a classic Fifo realized as ring buffer, a synchronized Fifo (`SyncFifo`), a Fifo for multiple readers (`MultipleReaderFifo`) and a Fifo to write or read continuous blocks (`BlockFifo`). Besides, a buffer for asynchronous data exchange has been implemented, providing always the latest data to the reader and the data not read will get lost (`CommBuffer`).

3.3 Network

LinkInterface

LinkInterface is a generic class to connect towards networks or hardware interfaces and to enable data transfer . It provides a unique access independent of the network type. The implementation is network dependent and MUST be done in derived classes specific to the platform.

```
Linkinterface:Linkinterface(long linkId)
```

A Link interface is identified by its ID, 0 is reserved for local broadcast messages. The id is used to determine from which link a message was received.

```
virtual void Linkinterface::init()
```

To initialize the hardware and establish physical interfaces, the `init()` function has to be called. Depending on the type of interface interrupt handlers are set (UART) or ports are selected (UDP) and, if possible, the local node number is initialized.

```
virtual bool Linkinterface::getNetworkMsg(NetworkMessage &inMsg,int32_t &numberOfReceivedBytes
↪ )
```

This function processes all received data. When a network message is complete, it returns true. However, if false is returned, the received message may contain incomplete data and should not be used.

```
virtual bool Linkinterface::isNetworkMsgSent()
```

If all buffered messages have been transmitted this function returns true. The next call to `sendNetworkMsg(...)` should then immediately start sending out the new messages.

```
virtual bool Linkinterface::sendNetworkMsg(NetworkMessage& outgoingMessage)
```

Gateway

A gateway is a link from the local middleware to an external network. A single middleware may have many gateways, connecting to one external link each. To send messages, a Gateway works like a normal subscriber to read data from the middleware topics, but it can respond to more than one topic. Messages received from the external link will be distributed in the loccal middleware using the same mechaninsm like normal topics. To read from the network link, the gateway implements a thread which may poll the link or may be reactivated by interrupts from the link. Some intelligent networks may deliver a list of topics which are expected externally. To access other networks, the user has to define which topics shall be forwarded (or forward all, but may be inefficient). The Gateway class inherits from `Subscriber` as well as `Thread`.

```
Gateway::Gateway(Linkinterface* linkinterface_, bool forwardall_ = false,bool enable_ = true
↪ );
```

A gateway is established passing an instance of a linking interface to either UDP or UART, the `forwardall_` flag decides whether all or only selected topics are forwarded , the `enable_` flag is set, when the gateway should send locally published topics to the network.

There are several ways to set the topics to be forwarded by the specified gateway:

```
void Gateway::setTopicsToForward(TopicListReport* topicList)
void Gateway::addTopicsToForward(TopicListReport* topicList)
void Gateway::addTopicsToForward(TopicInterface* topicId1,TopicInterface* topicId2=0,TopicInterface
↪ * topicId3=0,TopicInterface* topicId4=0)
void Gateway::resetTopicsToForward()
```

For simple networks, the list of topics which shall be forwarded to the network has to be known in the local node. This list has to be passed to the gateway using `setTopicsToForward(...)`

Intelligent networks (for example the implementation using UDP) the network provides this list and the receiver thread will call this method for arrived messages with topic 0 (list report).

```
void Gateway::setForwardAllTopics(bool forwardall = true)
```

In worst case all topics may be forwarded. A report of all topics to be forwarded by this gateway is received through:

```
TopicListReport* Gateway::getTopicsToForward()
```

Router

A router controls or connects several gateways at once. The class inherits from `Subscriber` and `Putter`. It is initialized with up to four gateways.

```
Router::Router(bool forwardTopicReports_ = false, Gateway* gateway1=0, Gateway* gateway2=0,
↪  Gateway* gateway3=0, Gateway* gateway4=0)
```

More gateways are added, if the maximum number of allowed gateways is not reached.

```
virtual void Router::addGateway(Gateway* gateway)
```

The network messages of all gateways are processed with `put(...)` (see Subscriber class), the messages to send out with `putGeneric(...)`(see Putter class).

Every packet that enters the router from local or the network is routed.
`virtual void Router::routeMsg(NetworkMessage &msg, long linkid)`

The default implementation uses the following functions therefore.
`virtual bool Router::shouldRouteThisMsg(NetworkMessage &msg, long linkid)`
`virtual bool RoutershouldRouteThisMsgToGateway(NetworkMessage &msg, long linkid,Gateway*`
↪ `gateway)`

UDP

RODOS offers different interfaces or wrappers for hardware specific implementations of the UDP port. Those are `UDPin` to receive data from a single UDP port, `UDPout` to send data, and `UdpInOut` for bidirectional communication.

The `UDPin` is opened with the port number on the local host used for reception.

```
UDPIn::UDPIn(const long portNr)
```

A UDP port may directly publish to a given topic. This topic however has to be accessed through `Subscriber::putFromInterrupt`.

```
UDPIn(const long portNr, Topic<GenericMsgRef>* associatedTopic)
```

The state of a UDP port is tested with two methods:
`bool UDPIn::readyToGet()` returns true, if incoming data is available, false otherwise.
`bool UDPIn::isConnected()` checks whether the port is active or not.

Data is received using two functions, one specifying a IP address the other not.

```
long UDPIn::get(void* userData, int maxLen)
long UDPIn::get(void* userData, int maxLen, unsigned long *ipaddr)
```

The outward port `UDPOut` requires a port number as well, and a hostname or IP Address (either in e.g. hex format, or consisting of separate octets).

```
UDPOut::UDPOut(const long portNr, const char* hostname = "localhost")
UDPOut::UDPOut(const long _portNr, unsigned long _ipAddr)
UDPOut::UDPOut(const long _portNr, int ip0, int ip1, int ip2, int ip3)
```

When sending Data also a IP address might be specified.

```
bool send(const void* userData, const int maxLen)
bool sendTo(const void* userData, const int maxLen, unsigned long ipAddr)
```

The class `UDPInOut` combines an In port and an Out port.

Chapter 4

Hardware Abstraction Layer

Abstract The RODOS HAL (Hardware Abstraction Layer) was developed to provide standardized access to buses like I2C and other functionalities like PWM provided by several platforms. The user can call the api-functions and does not have to think about the underlying implementation for the specific hardware.

4.1 API Definition

The API to send/receive messages from several kinds of buses is almost the same (write, read, writeRead), but the initialization/configuration of different kind of buses differs considerably. Therefore, even if the classes seem very similar, there are different classes for different hardware ports. These classes all inherit from one generic IO interface that defines the basic methods for accessing the buses or hardware functionalities (see figure 4.1).

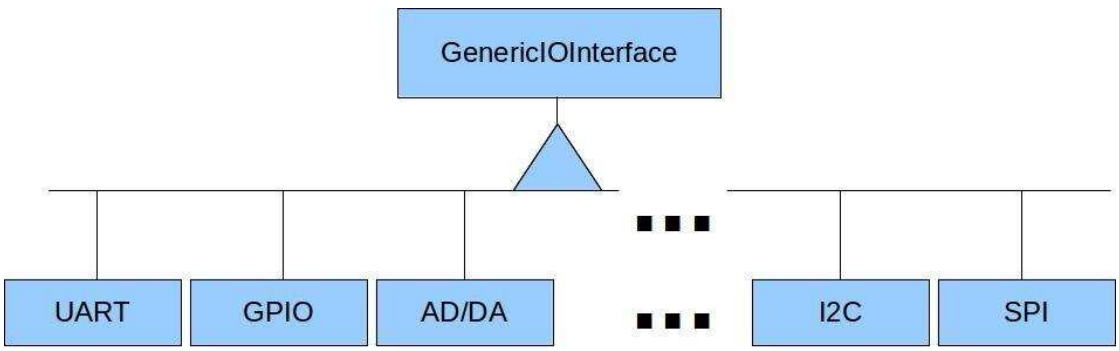


Figure 4.1: The GenericIOInterface is the base class for different hardware ports

4.1.1 Generic IO Interface

Listing 4.1: The generic IO Interface

```
1  class GenericIOInterface {
2  private:
3  IOEventReceiver* ioEventReceiver;
4  protected:
5  void* context;
6  public:
7  GenericIOInterface() {ioEventReceiver=0; }
8  /*** Implemented by each interface ****/
9  virtual int init(...)
10 {return 0;}
11 virtual void reset() { }
12 virtual int config(...) {return 0;}
13 virtual int status(...) {return 0;}
14 virtual bool readyToSend() {return false; }
15 virtual bool dataReady()
16 {return false; }
17 virtual int write(const char* sendBuf, int len) {return 0;}
18 virtual int read(char* recBuf, int maxlen) {return 0;}
19 virtual int writeRead(const char* sendBuf, int len, char* recBuf, int
       ↪ maxlen) {return 0;}
20 };
21 virtual void upCallSendReady() {if(ioEventReceiver) ioEventReceiver->
       ↪ SendFinished();}
22 virtual void upCallDataReady() {if(ioEventReceiver) ioEventReceiver->
       ↪ DataReady();}
```

The basic methods for accessing a bus can be divided into four groups: Initialization, current status, write/read and events after an action has finished (listing 4.1).

For initialization there are three methods. The `init()` and `config()` method will be found in each interface, but they all differ in parameter and execution. They are defines as virtual, just to signalize the subclasses will redefine it. Do not call it as virtual (from a pointer)! We recommend to do not configure the devices using the constructor but to use the init or config function. Many configs have as parameter the desired speed (baudrate). The driver will try to set the closest possible speed supported by hardware. Config returns a value ¡ 0 if an error occurred, else return 0 or a positive value (implementation dependent). The `reset()` method is used to reset the state of the bus.

Additionally, three methods can be used to retrieve the current status. The `status()` method returns the status for a given status type dependent of the bus implementation. The `readyToSend` ↪ `()` and `dataReady()`, per default return false, but each implementation shall return true if following calls to read or write would succeed.

The methods `write()`, `read()` and `writeRead()` transfer several bytes as a unit. Some implementations may do the transfer writing to FIFOS, other using a DMA and other by coping the data to an internal buffer and then send byte by byte triggered by interrupts. While a transfer is running (device busy), the corresponding function `readyToSend()` or `dataReady()` shall return

false. When a transfer is concluded, the driver shall call the corresponding `upCallSendReady()` or `upCallDataReady()` function and `readyToSend()` or `dataReady()` may return true again. The method `writeRead()` is a special case for bidirectional buses which are controlled by a single master, like for example UART RS-485, I2C, OneWire. For such buses the software has to signalize the hardware to toggle from write to read intermediately after the last transmitted bit. After this, data will arrive (from external device) and a buffer has to be ready to store the data. This is the second part of the parameter of `writeRead()`.

The functions `write()`, `read()` and `writeRead()` return in normal case the number of transferred bytes (`writeRead()` returns the sum of send plus received bytes). If the port is not ready and no transfer can be done they return 0. In error case they return negative values. The meaning of negative error numbers is not defined here and is implementation dependent.

The method `setIoEventReceiver()` is used to define an event receiver that will be called when an action has completed. In order to raise an event, the methods `upCallSendReady()` and `upCallDataReady()` have to be used.

All IO interfaces contain a pointer in their platform independent header file to a platform dependent class (e.g HAL_UART contains `HW_HAL_UART* context`). This class is only forward declared in the header file. The system programmer defines this class in his platform dependent implementation and puts hist data structures and methods there. The memory for this class should be allocated with xmalloc. If a proper initialization and constructor call is required the placement new operator can be used to construct an object on previously xmalloced memory. The context pointer in Genericiointerface is deprecated and shoudn't be used anymore.

4.2 Specific Interface API

All the IO interfaces that inherit from GenericIOInterface are located in `RODOSROOT/api/hal`. In these interfaces only the needed functions are implemented and sometimes additional methods have been added. Aside from the abstract interface for the specific port there is a generic class defined. This class is used as the type of the context and has no relevance for the end-user. This class is only relevant for the system developer to define this class for the specific needs of the corresponding hardware. Besides that, there are indexes defined to use multiple interface of the same type, for instance two I2C interfaces, on one controller.

4.2.1 GPIO(General Purpose Input Output) Pins

Listing 4.2: HAL_GPIO interface

```
1  // Generic class for hardware-specific context
2  class HW_HAL_GPIO;
3  // GPIO HAL
4  class HAL_GPIO : public GenericIOInterface {
5  private:
6  HW_HAL_GPIO *context;
7  public:
8  HAL_GPIO(GPIO_PIN pinIdx = GPIO_000);
9  int32_t init(bool isOutput = false, uint32_t numOfPins = 1, uint32_t
       ↪ initVal = 0x00);
10 void reset();
11 int32_t config(GPIO_CFG_TYPE type, uint32_t paramVal);
12 bool readyToSend() {return true;}
13 bool dataReady() {return true;}
14 void setPins(uint32_t);
15 uint32_t readPins();
16 void interruptEnable(bool enable = true,
17 GPIO_IRQ_SENSITIVITY mode = GPIO_IRQ_SENS_BOTH, void (*isr)() = NULL);
18 };
```

The GPIO HAL (listing 4.2) is designed to access one or more pins simultaneously. But be careful when defining pin-groups using "numOfPins" in init(). The use of all functions is limited to port boundaries! When port boundaries are exceeded only pins up to the most significant pin of the port will be set/read. Via the `interruptEnable()` function it is possible to react on changes to the pin. A proper service routine can be defined via this function. Instead of the read and write functions defined in GenericIOInterface the GPIO HAL uses the specific functions `setPins()` and `readPins()`.

4.2.2 PWM

Listing 4.3: HAL_PWM

```
// Generic class for hardware-specific context
class HW_HAL_PWM;
// PWM HAL
class HAL_PWM : public GenericIOInterface {
HW_HAL_PWM* context;
public:
HAL_PWM(PWM_IDX idx);
int init(int frequency = 1000, int resolution = 8);
int config(PWM_PARAMETER_TYPE type, int paramVal = 0);
void reset();
/*** only for PWM **/
int write(unsigned int pulseWidth);
};
```

The PWM HAL(listing 4.3) can be used to generate a PWM signal on a pin defined with the PWM_IDX. This HAL does not use the read/write functions but has a specific `write()` function

4.2.3 I^2C

Listing 4.4: HAL_I2C

```
// Generic class for hardware-specific context
class HW_HAL_I2C;
// I2C HAL
class HAL_I2C : public GenericIOInterface {
HW_HAL_I2C *context;
public:
HAL_I2C(I2C_IDX idx);
int32_t init(uint32_t speed = 400000);
void reset();
bool readyToSend();
bool dataReady();
};
int32_t write(const uint8_t addr, const uint8_t *txBuf, uint32_t
    ↪ txBufSize);
int32_t read(const uint8_t addr, uint8_t *rxBuf, uint32_t rxBufSize);
//
int32_t writeRead(const uint8_t addr, const uint8_t *txBuf,
uint32_t txBufSize, uint8_t *rxBuf, uint32_t rxBufSize);
```

The I^2C (listing 4.10) is developed to send and receive data vi the I^2C bus. To send and receive data without a STOP condition between the actions the `writeRead()` function can be used.

4.2.4 UART

Listing 4.5: HAL_UART

```
1  // Generic class for hardware-specific context
2  class HW_HAL_UART;
3  // UART HAL
4  class HAL_UART : public GenericIOInterface {
5  private:
6  HW_HAL_UART* context;
7  public:
8  HAL_UART(UART_IDX uartIdx);
9  /* Initialization of uart interface: mode 8N1, no HW flow control*/
10 int init(unsigned int baudrate = 115200);
11 void reset();
12 int config(UART_PARAMETER_TYPE type, int paramVal);
13 int status(UART_STATUS_TYPE type);
14 bool readyToSend();
15 bool dataReady();
16 int write(const char* sendBuf, int len);
17 int read(char* recBuf, int maxLen);
18 };
19 int getcharNoWait();
20 int putcharNoWait(char c);
21 // returns character on success else -1
22 // returns c on success else -1
```

The UART HAL has two specific functions for sending and receiving a single character via the UART bus. The `config()` function accepts the folowing parameters:

- `UART_PARAMETER_BAUDRATE`,
- `UART_PARAMETER_HW_FLOW_CONTROL`,
- `UART_ENABLE_DMA`

When a specific feature is not supported by the hardware -1 is returned.

4.2.5 CAN

Listing 4.6: HAL_CAN

```
1  // Generic class for hardware-specific context
2  class HW_HAL_CAN;
3  // CAN HAL
4  class HAL_CAN : public GenericIOInterface {
5  private:
6  HW_HAL_CAN* context;
7  public:
8  HAL_CAN(CAN_IDX canIdx);
9  int init(unsigned int baudrate);
10 void reset();
11 int config(CAN_PARAMETER_TYPE type, int paramVal);
12 int status(CAN_STATUS_TYPE type);
13 bool readyToSend();
14 bool dataReady();
15 };
16 /*** only for CAN **/
17 // Add a filter for incoming Messages. Only Message which pass a filter
       ↪  are received.
18 // Filters may be implemented in hardware an can be limited.
19 bool addIncomingFilter(uint32_t ID, uint32_t IDMask=0, bool extID =
       ↪ true,
20 bool rtr=false);
21 int write(const char* sendBuf, int len,uint32_t canID, bool extID =
       ↪ true,
22 bool rtr=false);
23 int read(char* recBuf, uint32_t* canID=0, bool* isExtID = 0,bool* rtr
       ↪ =0);
```

The CAN HAL does not use the standard read/write functions but has own functions for reading and writing data to the CAN bus (listing 4.6). Every call to read/write receives/sends exactly one CAN frame. So there can be be maximal 8 bytes received/transmitted in one call. Besides that, there is a `addIncomingFilter()` function to specify which CAN Frames should be received. Frames not matching a filter may be already dropped by the controller hardware. It is possible to create multiple instances of `HAL_CAN` for the same CAN controller to distribute CAN messages to different applications/threads. If the identifier of an incoming frame matches the filter of two or more `HAL_CAN` objects the result is undefined.

4.3 Using of a HAL-interface

The usage of the HAL is very simple as shown in the following minimal example (listing 4.7). This will work fine if the I^2C port is used in one thread only:

Listing 4.7: SensorReader

```
#include "rodos.h"
HAL_I2C i2cBus(I2C_IDX1); //static instance of HAL
class SensorReader: public Thread {
void run() {
i2cBus.init(); //one time initialisation of this bus
uint8_t sensorValue[2];
TIME_LOOP(0,1*SECONDS)
{
i2cBus.read(0x30, sensorValue, 2); //several times reading the bus
}
}
} sensorReader;
```

There are a few rules to follow to get everything working properly:

- It is recommended to allocate all drivers statically (as shown in the example above), no new and not on the stack.
- If several threads are using the same bus, e.g. I2C port 1, only one instance for all threads should be created. All the threads should use the instance, but protect their access with semaphores. (exception: HAL_CAN, there you can create multiple inteances for the same CAN controller and it is internally protected with semaphores) To initiate a bus that is used by several threads is important to use an Initiator as shown in the example below, otherwise it cannot be guaranteed that the thread, that runs the bus initialization, runs first. An initialization in every thread will cause even more problems.
- The `config()` method (when used) has to be executed after the `init()` method.
- Implement everything as simple as possible.

Following, there is another example showing the use of an Initiator (listing 4.8). This Initiator will be executed before the threads start running.

Listing 4.8: SensorInitiator

```
#include "rodos.h"
HAL_I2C i2cBus(I2C_IDX1); //static variable
static class sensorInitiator : public Initiator {
void init() {
//runs before the threads start running
i2cBus.init(); //one time initialisation of this bus
}
} sensorInitiator;
class Adxl1Reader1: public Thread {
void run() {
uint8_t adxl1Value[2];
TIME_LOOP(0,1*SECONDS)
{
i2cBus.read(0x30, adxl1Value, 2); //several times reading the bus
}
}
} adxl1Reader1;
class GyroReader2: public Thread {
void run() {
uint8_t gyroValue[2];
TIME_LOOP(0,2*SECONDS)
{
i2cBus.read(0x32, gyroValue, 2); //several times reading the bus
}
}
} gyroReader2;
```

4.4 Implementation of a HAL-interface for a new hardware platform

To create a HAL for a new hardware platform it is necessary to implement all the functions defined in the interface in api/hal. This implementation has to be placed in the source folder for the corresponding port in a separate hal folder, e.g. `src/bare-metal/stm32f4/hal/`. For saving the platform specific context of each I/O interface, the generic class defined in the api (e.g. HW_HAL_I2C) has to be implemented. In this class all hardware specific data and functions shall be defined. The following shows an example of such a hardware specific context class implementation with attributes for the I/O pins and specific functions:

Listing 4.9: HW_HAL_I2C

```
class HW_HAL_I2C {
public:
HW_HAL_I2C(I2C_IDX i2cIdx){I2C_idx = i2cIdx;}
HW_HAL_I2C(){};
I2C_IDX I2C_idx;
uint16_t I2C_SCL_PIN;
uint16_t I2C_SDA_PIN;
...
};
int32_t sendAddr(const uint8_t addr, uint8_t rwFlag){
//send the adress implementation };
int32_t start(){ //send start signal implementation };
int32_t stop(){ //send the adress implementation };
```

A static array of type `HW_HAL_I2C` has to be created to allocate memory for the contexts for every I2C port:

```
HW\_HAL\_I2C I2C\_contextArray[I2C\_IDX\_MAX-1];
```

If the HW_xxx class contains large buffers it may be a better idea to allocate the memory for it with xmalloc/placement new in the constructor of the corresponding HAL_xxx object. If the definition of the `HW_HAL_I2C` is done, the API functions can be implemented. The next example is the constructor implementation. In this constructor the context shall be initialized. No hardware initialization should be done here because we do not have any control when each constructor will be executed and in which sequence.

Listing 4.10: HAL_I2C::HAL_I2C

```
HAL_I2C::HAL_I2C(I2C_IDX idx) {
// placement new to avoid dynamic memory allocation
context = new (&I2C_contextArray[idx - 1]) HW_HAL_I2C(idx);
switch (idx) {
case I2C_IDX1:
context->I2Cx = I2C1;
break;
case I2C_IDX2:
...
}
}
```

In the `init()` method the hardware should be initialized with the given parameters e.g. speed:

Listing 4.11: HAL_I2C::init

```
int32_t HAL_I2C::init(uint32_t speed) {
context->I2C_SPEED = speed;
/* enable peripheral clock for I2C module */
RCC_APB1PeriphClockCmd(context->I2C_CLK, ENABLE);
...
}
```

To implement the `write()` and `read()` functions it is recommended to split the needed code in smaller functions in the context class and call them. For example:

Listing 4.12: HAL_I2C::write

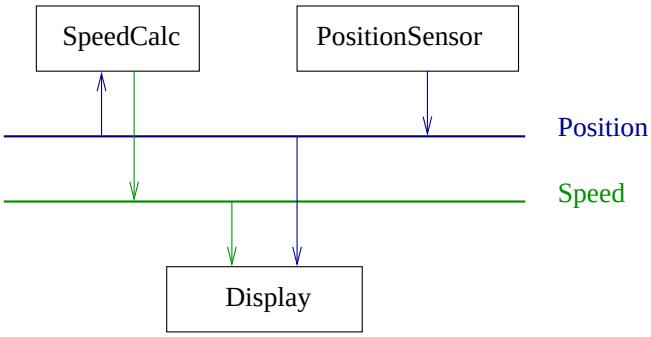
```
int32_t HAL_I2C::write(uint8_t addr, uint8_t* txBuf, uint32_t txBufSize
    ↪ ) {
context->start();
context->sendAddr(addr, WRITE);
context->write(txBuf,txBufSize);
context->stop();
return SUCCESS;
}
```

Chapter 5

Examples

5.1 GPS

The following example includes topics with more than one subscribers and of subscribers of more than one topic. Source files are found in folder `rodos-tutorials/first-steps/gps`. A position sensor measures and publishes data of the postion (3D) of a flying object. A speedcalculator receives those data and calculates and publishes the object's speed. Finally, a display subscribing both topics, position and speed, prints the data.



All in all, the application consists of five different files and is therefore a good example for the modular character of RODOS applications. The two topics `Topic<Pos> position` and `Topic<double> speed` are declared in `topics.cpp` and included through `topics.h`, which also defines a threedimensional position in the structure `Pos`.

Each application has its own file: `positionsensor.cpp`, `speedcalc.cpp` and `display.cpp`. Below those files are described in details.

positionsensor.cpp

This file defines the position sensor. It is a simple thread which acts as a publisher. To emulate the measurement, a local `Pos` variable is updated with random values and published to the

`position` topic.

Listing 5.1: positionsensor.cpp

```
#include "rodos.h"
#include "topics.h"
#include "random.h"

class PositionSensor : public Thread {
public:
    PositionSensor() : Thread("PositionSensor") { }

    void run () {
        Pos p = {0.0, 0.0, 0.0};
        TIME_LOOP(2*SECONDS, 3*SECONDS) {
            p.x+= (randomTT800Positive() % 40)*0.05-1;
            p.y+= (randomTT800Positive() % 40)*0.05-1;
            p.z+= (randomTT800Positive() % 40)*0.05-1;
            position.publish(p);
        }
    }
} positionSensor;
```

Of course, the RODOS header and the topic declaration has to be included, as well as `random.h` in order to generate the random numbers.

speedcalc.cpp

The speed calculator is the subscriber corresponding to the position sensor publisher. The class `SpeedCalc` inherits from `Subscriber` and has therefore to override the `put(...)` method.

Listing 5.2: speedcalc.cpp

```
#include "rodos.h"
#include "topics.h"
#include "math.h"

class SpeedCalc :  public Subscriber {
public:
    SpeedCalc() : Subscriber(position, "SpeedCalc") { }
    Pos p0,p1;
    long put(const long topicId, const long len, const void* data,
        ↪ const NetMsgInfo& netMsgInfo) {
        p0=p1;
        p1=*(Pos*)data;
        double v = sqrt((p0.x-p1.x)*(p0.x-p1.x)+(p0.y-p1.y)*(p0.y-p1.y)
            ↪ +(p0.z-p1.z)*(p0.z-p1.z));
        speed.publish(v);
        return 1;
    }
} speedCalc;
```

`SpeedCalc` has two attributes of type `Pos`. This is necessary, as the speed is derived from the position at time t_n and at time t_{n-1}. The `put(...)` method does not store the received position value, but instead uses it to calculate the velocity and to publish it to topic `speed`.

display.cpp

The `Display` thread subscribes to both, speed and position, and prints their values to the screen. Therefore it declares two subscribers, one for each topic, both initialized with their own putters, in this case a `CommBuffer`.

Listing 5.3: display.cpp

```
1  #include "rodos.h"
2  #include "topics.h"

4  static CommBuffer<Pos>    posbuf;
5  static CommBuffer<double> speedbuf;
6  static Subscriber namenotimportant1(position, posbuf,   "posreceiverbuf
       ↪ ");
7  static Subscriber namenotimportant2(speed,    speedbuf, "
       ↪ speedreceiverbuf");

9  class Display : public Thread {
10     void run () {
11         TIME_LOOP(1*SECONDS, 1*SECONDS) {
12             Pos p;
13             double v;
14             posbuf.get(p);
15             speedbuf.get(v);
16             PRINTF( "Position (%3.2f;%3.2f;%3.2f) speed %3.2f\n",p.x, p
                   ↪ .y, p.z, v);
17         }
18     }
19 } display;
```

5.2 Task Distribution

The task distribution example shows how to distribute tasks in different computer nodes and how to control, on which computer node a task shall be executed. This strategy can be used for both: fault tolerance and load balancing. For source files please have a look into folder:
`rodos-tutorials/advanced_and_complex/task_distribution-central/`

The job of the system is performed by *user tasks*, which are supported by task managers and a task distributor. There may be different types of user tasks. Tasks of the same type (on different nodes) interchange their context periodically, to allow a sleeping task to start working with the most recent context.

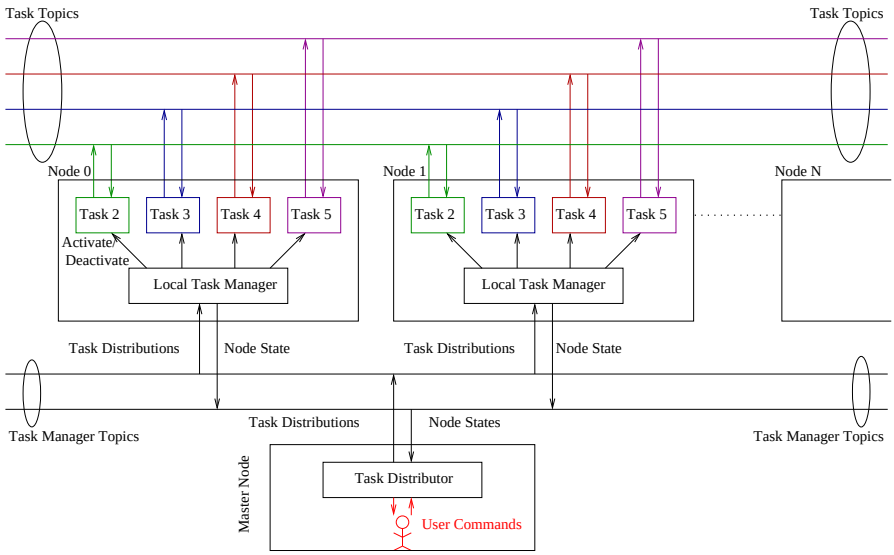


Figure 5.1: Overview of the task distribution

Each node has a local task manager which controls which tasks shall be active and which shall be sleeping (not active) in the local node. The local task manager periodically sends state information of the own node to a central task distributor.

The task distributor decides on which node which tasks shall be executed. If a node crashes, the distributor will command local task managers to migrate the tasks of the crashed node. To migrate a task X from node A to B means to deactivate (set to sleep) the task X in node A and activate another instance of the task X task in node B. If the node A fails, then we just have to activate task X in B. The task distributor may be executed in any node, together with other tasks, but for this example it will be executed on its own node, which is called the master node. A more elaborated implementation could distribute and/or replicate the task distributor too, but for this example, we take a simple solution.

All nodes shall have a local task manager and all user tasks loaded. The number of nodes is not fixed. Each node (more exactly: the local task manager on each node) will report it's state to the task distributor. So the task distributor knows which nodes are ready to execute tasks.

The task distributor creates a task distribution table and distributes it to all local task managers. Each local task manager can see on this table which tasks shall be active in it's node. An overview of the example is shown in figure 5.1.

Each node has exactly one instance of each user task type. All tasks of the same type (distributed in several nodes) interchange their context using a dedicated topic of the middleware.

The local task manager uses the application discovery protocol to find the loaded task in it's own node, and to access the application interface of the loaded tasks. The application interface provides **activate()** and **deactivate()** methods.

The local task managers hear (subscribers) from the task distribution topic to get the distribution table to know which tasks shall be active and which shall be sleeping in their node.

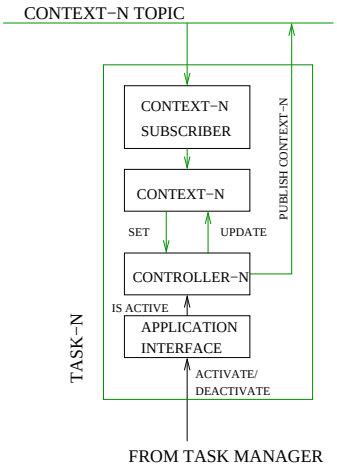


Figure 5.2: Structure of a user task

The local task manager reports the state of it's node using the node state topic. The task distributor hears (subscriber) messages from the node state topic to know which nodes are available. Using this informations it builds the task distribution table and distributes this table using the topic task distribution.

All topics are declared in the `globals.cpp` file.

Listing 5.4: used topics

```
1  /***************** Topics (topicId -1 -> automatic generated from name)
       ↪  *****/

3  Topic<DistributionTable> distributionTableTopic(-1, "
       ↪ distributionTableTopic");
4  Topic<NodeState>         nodeStateTopic(-1, "nodeStateTopic");

6  Topic<ContextNegCnt>  negCntTopic(-1, "negCntTopic");
7  Topic<ContextOddCnt>  oddCntTopic(-1, "oddCntTopic");
8  Topic<ContextEvenCnt> evenCntTopic(-1, "evenCntTopic");
9  Topic<ContextPrimCnt> primCntTopic(-1, "primCntTopic");
```

In this example all user tasks have the same structure shown in figure 5.2 (to make it simpler) which is recommended as starting point for real applications. We will have a look at one of the user task implementations for detail, the `EvenCntController` (listings 5.5 to 5.7), which task is - as its name says - to count even numbers.

Listing 5.5: usertask_evencnt.cpp 1st Part

```
1    static Application evenCnt("evenCnt", TASK_EVENCNT);
2    static CommBuffer<ContextEvenCnt> contextEvenCnt;
3    static Subscriber nameNotImportant(evenCntTopic, contextEvenCnt);
```

Listing 5.5 contains three lines. The first line declares the user task as an application with name and id, to make it possible for the task manager to find it. Each user tasks, as well as the task manager and task distributor are building blocks declared as applications. A central point of each user task is it's context. In the examples the context is just a simple counter (long long). The context is used by the controller – when it is active – to compute a behaviour. Each time the controller modifies the context it distributes the updated version using the corresponding topic. In this case the topic is called `evenCntTopic`. The subscriber of all other tasks of the same type (on other nodes) will get a copy of the current context and update their local context. So when a task is activated (in any node) it can take the most recent context to continue operations.

Listing 5.6: usertask_evencnt.cpp 2nd Part

```
1    class EvenCntController : public Thread {
2    public:
3       EvenCntController() : Thread("EvenCntController") { }
4       void run () {
5      ContextEvenCnt myCnt;
6            AT(1*SECONDS); PRINTF("%s%d\n", EC, TASK_EVENCNT);

8      evenCnt.isActivated = false; // wait orders from taskmanager

10     TIME_LOOP(300*MILLISECONDS, 500*MILLISECONDS) {
11     .
12     .
13     .
14     }
15   } evenCntController
```

Listing 5.7: usertask_evencnt.cpp 3rd Part

```
1  TIME_LOOP(300*MILLISECONDS, 500*MILLISECONDS) {
2      if(!evenCnt.isActivated) { // still sleeping
3        PRINTF("%s-------\n",EVEN);
4      continue;
5      }

7      contextEvenCnt.get(myCnt);
8      myCnt.cnt += 2;

10     PRINTF("%s%7lld\n", EVEN, myCnt.cnt); // \n just to force fflush
11     evenCntTopic.publish(myCnt);
12   }
```

On the lower part of each task, we have the application interface which gets commands – activate / deactivate – from the local task manager. The application interface just sets a flag to indicate if the task shall be active or sleeping. The controller checks this flag to know if it shall work or sleep.

The local task manager (see figure 5.4) creates a list of loaded applications (tasks) on the local node. For this purpose it uses the standard application discovery protocol of the RODOS appli-

cation interface. The task manager uses this list to send activate/deactivate commands to each application using the standard application interface.

Listing 5.8: Using the application interface to find loaded applications

```
/***** applications discovery **/
    for(int i = 1; i < NUM_OF_TASKS; i++) {
        taskReferences[i] = Application::findApplication(i);
    }
```

The task manager has a subscriber which gets the task distribution table from the task distribution topic. The subscriber updates the local distribution table. The task manager core checks this table to activate the tasks which shall be running in the current node and deactivate all other tasks.

Listing 5.9: Activation and deactivation of user tasks

```
for(int i = FIRST_USERTASK; i < NUM_OF_TASKS; i++) {

          if(myDistributionTable.nodeToExecute[i]  == nodeNumber) {
        numOfTasks++;
          if(taskReferences[i]) taskReferences[i]->activate();
        } else {
          if(taskReferences[i]) taskReferences[i]->deactivate();
        }
```

The task manager core periodically collects state information of the local node, in this example just an indicator for CPU load. Other possibilities could be: free memory, list of loaded tasks, etc. The task manager publishes the node state using the node state topic.

Listing 5.10: Updating the node state

```
 /** Report me: nodeState **/
      nodeState.reportTime = NOW();
      nodeState.nodeSane   = true;
      nodeState.cpuLoad    = numOfTasks;
      nodeStateTopic.publish(nodeState);
      PRINTF("%s%ld%s%d%s%ld%s%5.3f\n",
          NR,   nodeState.nodeNumber,
          S,    nodeState.nodeSane,
          L,    nodeState.cpuLoad,
          TIME, (double)nodeState.reportTime / SECONDS)
```

The task distributor (see figure 5.3) has a subscriber from the node state topic, to know which nodes are usable. In this example all nodes are supposed to be identical, in more elaborated implementations nodes could have a different task list and may have different performance, they could have different devices attached, which would be reflected in the list of loaded tasks (interfaces to devices).

Knowing which nodes are available and executing user commands, the task distributor core creates a task distribution table which says, for each task, on which node it shall be executed.

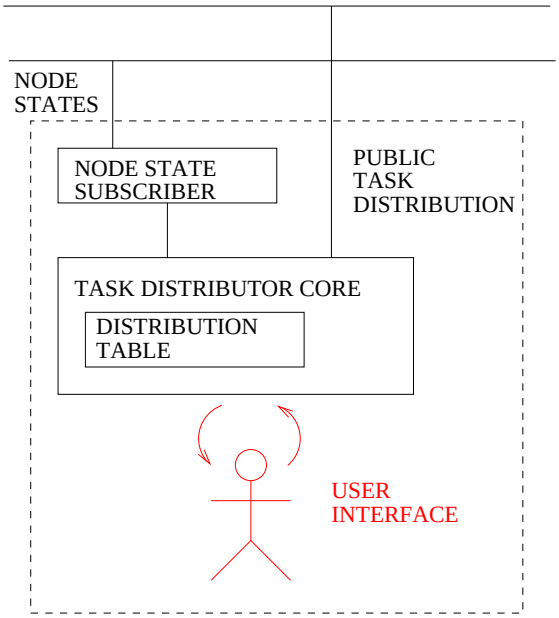


Figure 5.3: Task Distributor

The task distributor distributes (publishes) this table periodically using the task distribution topic.

To execute (on linux) first create the linux lib: Call in the rodos root directory

```
% source make/rodosenv
% linux-lib
```

In the directory where this example is stored (tutorials/taskdistribution) use the ready to use shell script

```
% executeit
```

This script will create a user boot image (an executable) for the user nodes and a distributor boot image for the master node. The user boot image contains all user applications and the task manager. The distributor boot image for the master node which contains only the task distributor. Then the script will create 4 xterm windows, one for each user node, one xterm window for the task distributor and one xterm window for the network simulator. Note: The taskdistributor (yellow window) will wait until you type <c> to start working. Each xterm window will start the corresponding image. Try closing user windows and see how the distributor migrates the tasks in the still running nodes. Try typing commands to the distributor. Syntax: <application-Nr> <node-Nr> no names, no commas, just two numbers separated by blank. The first number is the application number (from 2 to 5), the second number is the node where it shall be executed. -1 means no node, task is nowhere active. Die task distributor will distribute the task in such a way that the maximal load (number of tasks) of any node shall not be bigger than minimal load + 1. If a node crashes (try control-C on any user node) the tasks from the crashed nod will be distributed on the oner nodes. If a new node comes (create a new window and execute “tst_usernode”) the distributor will give it some tasks to execute.

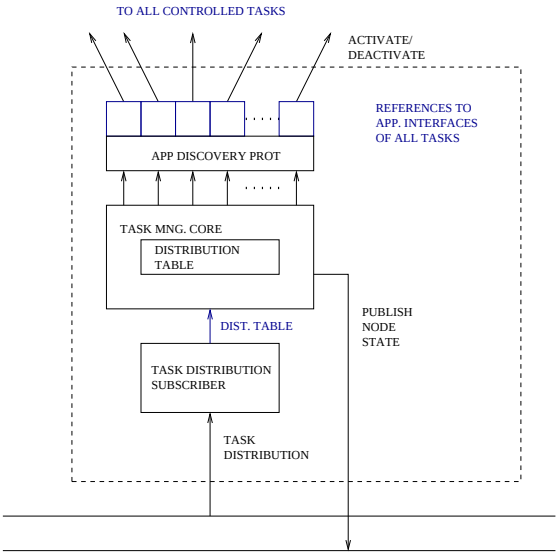


Figure 5.4: Local task manager