How to get started with RODOS in 9 easy steps

RODOS Tutorial

Version:	**1.0**
Document Id:	**Tutorials-1**
Date:	**03.06.2013**
Author:	**Johannes Freitag**

0. Before start

0.1. Notes on this tutorials

- A “$” means that the following command has to be executed in a terminal

0.2. RODOS directory structure

- **make**
  contains build scripts - with these scripts it is possible to compile RODOS applications for a variety of hardware platforms
- **api**
  all header files defining the Application Programming Interface – have a look at these files to see all possible RODOS functions
- **tutorials**
  learn how to use RODOS
- **doc**
  more documentation
- **src**
  all RODOS core source files - not important for the RODOS users

0.3. Steps to compile and execute a RODOS program

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.

1. Hello World

The Hello World tutorial is the most simple RODOS program. It only prints the string “Hello World!” in one thread.

1.1. Used RODOS functions

- **PRINTF( )**
  basically the same as the standard C printf() function – prints characters and numbers to terminal

1.2. Program helloworld.cpp

```
#include "rodos.h"

class HelloWorld : public Thread {

  void run(){
    PRINTF("Hello World!\n");
  }
} helloworld;
```

includes the RODOS API

defines a Thread named HelloWorld

implements the code for the main task of the thread
In this case: print “Hello World!”

instantiates one thread of type HelloWorld

1.3. Compiling

Compile the tutorial as described in chapter 2, in the following steps:

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 hello world tutorial
   $ cd tutorials/first-steps
6. Compile the user program
   $ linux-executable helloworld.cpp
7. Execute the binary
   $ ./tst
8. Exit the program with Ctrl+C

1.4. Console output

After some RODOS Information:

```
--------------- application running ------------
Hello World!
```

printout of run() method

1.5. Several Threads

Try now helloworld-multiple.cpp. Do you see the diffeence?

2. Basic structure

The Basic structure tutorial is an extension to the Hello World tutorial. It prints the string “Hello World!” and implements the basic structure of a RODOS program consistent of one application and one thread.

2.1. Used RODOS functions

- **PRINTF( )**
  basically the same as the standard C printf() function – prints characters and numbers to terminal

2.2. Program basic.cpp

includes the RODOS API

```
#include "rodos.h"
```

application that wraps all threads, events,... in this file

```
static Application appHW("HelloWorld");
```

defines a thread named HelloWorld

```
class HelloWorld : public Thread {

public:
```

thread constructor with definition of the thread name

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

is called before the scheduler starts

```
  void init() {
    PRINTF("Printing Hello World");
  }
```

implements the code for the main task of the thread
In this case: print “Hello World!”

```
  void run(){
    PRINTF("Hello World!\n");
  }
}
```

instantiates one thread of type HelloWorld

```
static HelloWorld helloworld;
```

2.3. Compiling

Compile the tutorial as described in chapter 2, in the following steps:

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 hello world tutorial
   $ cd tutorials/first-steps
6. Compile the user program
   $ linux-executable basic.cpp
7. Execute the binary
   $ ./tst

8. Exit the program with Ctrl+C
9. Modify the run() method in basic.cpp
10. Repeat step 6 to 8 and see the difference

2.4. Console output

```
RODOS RODOS-100.0 OS Version RODOS-linux-8
Loaded Applications:
         10 -> 'Topics & Middleware'
       1000 -> 'HelloWorld'
Calling Initiators
Distribute Subscribers to Topics
List of Middleware Topics:
 CharInput  Id = 28449 len = 12.   -- Subscribers:
 SigTermInterrupt  Id = 16716 len = 4.   -- Subscribers:
 UartInterrupt  Id = 15678 len = 4.   -- Subscribers:
 TimerInterrupt  Id = 25697 len = 4.   -- Subscribers:
 gatewayTopic  Id = 0 len = 12.   -- Subscribers:

Event servers:
Threads in System:
   Prio =       0 Stack =  32000 IdleThread: yields all the time
   Prio =     100 Stack =  32000 HelloWorld: Printing Hello World
BigEndianity = 0, cpu-Arc = x86, Basis-Os = baremetal, Cpu-Speed (K-
Loops/sec) = 350000
-----------------------------------------------------
Default internal MAIN
--------------- application running ------------
Hello World!
```

RODOS version

all applications in this programm

shows defined threads and printout of init method

printout of run() method

3. Time

This tutorial 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.**

3.1. RODOS time functions

- **NOW()**
  Returns the current time (in nanoseconds)
- **SECONDS_NOW()**
  Returns the current time in seconds
- **AT(time)**
  Suspends (interrupts) the thread that has called this method, until the given point in time is reached
- **TIME_LOOP(firstExecution, Period) { … }**
  Almost each control loop has a start time and a period. This macro provides this loop with no end.

3.2. Time macros

In order to use the time functions comfortably there are some time macros defined: NANOSECONDS , MICROSECONDS , MILLISECONDS , SECONDS , MINUTES, HOURS , DAYS , WEEKS , END_OF_TIME

To use them, just multiply them to the amount of time, e.g. AT(3*SECONDS) . END_OF_TIME is the highest time possible (about 293 years).

3.3. Program time.cpp

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

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

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

waits for the point in time: 3 seconds after start

waits for 1 second

code in the loop will be executed every 2 seconds; the first execution will be at 5 seconds after start

3.4. Compiling and console output

Compile the tutorial time.cpp in tutorials/first-steps as described in chapter 2 and execute it. The output should be the following:

```
waiting until 3rd second after start
after 3rd second
waiting until 1 second has pased
1 second has pased
print every 2 seconds, start at 5 seconds
current time:   5.000003995
current time:   7.000004191 ...
```

4. Priority

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.

In this tutorial 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 “*”) runs when one second is over, although the low priority thread (printing “.”) does not suspend.

The priority of a thread is defined in the thread constructor.

4.1. Program priority.cpp

```
...
HighPriorityThread() : Thread("HiPriority", 25)
...
LowPriorityThread() : Thread("LowPriority", 10)
...
```

thread with priority 25

thread with priority 10

4.2. Compiling and console output

Compile the tutorial priority.cpp in tutorials/first-steps as described in chapter 2 and execute it. The output should be the 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 ------------
*..........*..........*..........*...........*..........*....
```

low priority thread

interrupted by high prio thread

Switch the priorities of the threads, compile again and see the difference.

4.3. Special function: Priority ceiling

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 fist it is the same as the priority.cpp example but after leaving the fist 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 chapter 2 and execute it. The output should be the following:

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

no more interrupts from the high prio thread

5. Thread Communication

The communication between two threads can bee realized via a CommBuffer or a FiFo (First in first out). For that a CommBuffer or a Fifo has to be defined outside a thread so that both threads can access it.

5.1. CommBuffer

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. Not using a CommBuffer is risky, because maybe the data is half written in the shared variable while the thread is interrupted. In this case the receiver thread gets inconsistent data.

5.1.1. Programm combuffer.cpp

```
CommBuffer<int> buf;
```

CommBuffer of type Integer

```
class Sender : public Thread {
...
  PRINTF("Writing %d\n", cnt);
  buf.put(cnt);
...
}
```

thread puts local counter data into the CommBuffer

```
class Receiver : public Thread {
...
  buf.get(cnt);
  PRINTF("Reading %d\n", cnt);
...
}
```

thread gets counter data from the CommBuffer and saves it into local variable

Compile the tutorial combuffer.cpp in tutorials/first-steps as described in chapter 2 and execute it.

5.2. Fifo

A fifo is used for synchronous communication from one single writer to one single reader. 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.

5.2.1. Programm fifo.cpp

```
Fifo<int, 10> fifo;
```

Fifo for 10 Integer values

```
class Sender : public Thread {
...
      bool ok = fifo.put(cnt);
...
}
```

puts the current counter value into the fifo and checks whether the fifo is full

```
class Receiver : public Thread {
...
      bool ok = fifo.get(cnt);
...
}
```

receives the current counter value from the fifo and checks whether the fifo is empty

Compile the tutorial fifo.cpp in tutorials/first-steps as described in chapter 2 and execute it.

5.3. Synchronous Fifo

A SyncFifo is basically the same as Fifo, but in this case the sender will be suspended if the fifo is full and the receiver will be **suspended until data is ready**.

Compile the tutorial fifo_sync.cpp in tutorials/first-steps as described in chapter 2 and execute it.

5.4. Which is best for what?

If the receiver needs only the latest data a Commbuffer should be used. If the receiver needs all the data from the sender and in the right order, a Fifo is the way to do it. A SyncFifo is a good option if the data has to be processed short times after sending it, but take notice that the thread cannot do anything until new data is available.

6. Critical sections

To avoid concurrent access of critical sections semaphores have to be used. To enter a semaphore use sema.enter() and to leave use sema.leave(). I

6.1. RODOS functions

- **Semaphore::enter()**

  Makes a thread enter a semaphore. All other threads trying to enter the same semaphore will wait until it has been left again.

- **Semaphore::leave()**

  Leaves the semaphore and allows other threads entering it.

- **PROTECT_WITH_SEMAPHORE(sema){ … }**
  A macro entering the semaphore "sema" befor the surrounded code (critical section) and leaving it afterwards. It is only a short cut, which may be usefull .... or maybe not.

- **yield()**
  Interrupts the current thread and calls the scheduler that looks for a thread to execute. If no other thread wants to be executed, the thread continues.

6.2. Program semaphore.cpp

```
Semaphore onlyOne;
...
  onlyOne.enter();
  PRINTF(" only one, I am -- %02d -- ,", myId);
  yield();
  PRINTF("time %3.9f\n", SECONDS_NOW());
  onlyOne.leave();
...
```

semaphore definition outside the threads

enters semaphore “onlyone”

printout interrupted by yield, but because of the semaphore no other thread can print between the id and the time printout

leaves semaphore “onlyone”

6.3. Program semaphore_macro.cpp

```
The same functionality but using the macro short cut.
...
  PROTECT_WITH_SEMAPHORE(onlyOne){
    PRINTF(" only one, I am -- %02d -- ,", myId);
    yield();
    PRINTF("time %3.9f\n", SECONDS_NOW());
  }
...
```

protection with semaphore “onlyone”

end of the chritical section

6.4. Compiling and console output

Compile the tutorial semaphore.cpp in tutorials/first-steps as described in chapter 2 and execute it. The output should be the following:

```
 only one, I am -- A -- ,time   3.000056382
 only one, I am -- B -- ,time   3.000077366
 only one, I am -- C -- ,time   3.000094338
 only one, I am -- D -- ,time   3.000111110
 only one, I am ---E -- ,time   3.000128005
 only one, I am -- F -- ,time   3.000157338
 only one, I am -- G -- ,time   3.000180353
...
```

Remove the protection in semaphore.cpp, compile again and see the difference.

6.5. Attention: A deadlock may occur!

Compile and have a look at the tutorial semaphore_deadlock.cpp. The program will stop when a deadlock has occurred.

7. 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. Just use them to trigger threads that handle the interrupts. Implement them as short as possible.

An event has basically two methods: The init() method simmilliar to threads and the handle() method in which the code is defined that handles the event.

7.1. RODOS functions

- **activatePeriodic(startTime, period)**
  Activates an event periodically after the first activation at startTime.
- **activateAt(time)**
  Activates an event at the given point in time.
- **thread.resume()**
  Resumes a thread that is suspended.

7.2. program event.cpp

```
class TestWaiter: public Thread {
...
  PRINTF("Suspend and wait until some one resumes me\n");
  AT();            suspends the thread forever
  PRINTF("testwaiter running again at %3.9f\n", SECONDS_NOW());
...
}
                                             defines an event
class TimeEventTest : public TimeEvent {
public:
  void handle(){          handles the event
    xprintf("   Time Event at %3.9f\n", SECONDS_NOW());
    testWaiter.resume();       resumes the suspendend thread
    xprintf("   Testwaiter resumed from me\n");
  }

  void init() { activatePeriodic(5*SECONDS, 3*SECONDS); }
};
...                   defines when the event is beeing raised
                      could also use activateAt(time)
```

7.3. Compiling and console output

Compile the tutorial event.cpp in tutorials/first-steps as described in chapter 2 and execute it. The output should be the following:

```
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
...
```

Try the example with activateAt(time) instead of activatePeriodic(startTime, period).

8. Middleware

Up to here, we had "normal" programming. Now assume we are in a big team with a big project. You do not know the details of what others are programming, just the format of the data you need from them or you produce for them. Now you have to get and distribute this data without notion of the other side of this generic interface which we call the middleware.

The middleware is used to communicate between tasks and even between tasks of different RODOS nodes. This communication is based on a **publisher/subscriber protocol** and there is no connection from a sender to a receiver.

Any thread can publish messages under a given topic, while subscribers of the same topic receive the published data.

There can be 0, 1 or many publishers for one topic. The same goes for subscribers.

8.1. Required files

For this example you will need flowing files:

sender.cpp	The one who sends test message: a publisher
topics.h, topics.cpp	communication channels to send and receive data
receiver_*	different methods to subscribe and get data: Subscribers

You will need to compile several source files together. Every compilation needs to include the file topics.cpp

for example:

$linux-executable topics.cpp sender.cpp receiver_commbuff.cpp

or another receiver:

$linux-executable topics.cpp sender.cpp receiver_putter.cpp

or all receivers togehter:

$linux-executable topics.cpp sender.cpp receiver_*.cpp

8.2. Topic, program topics.cpp

A topic is a pair of a data-type and the topic id, e.g.:

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

name of the topic

data type of published messages

topic id

If the topic id is “-1” the id will be generated.

8.2.1. Sample topics

Some sample topics are defined in topics.cpp. To use these topics in a RODOS program include topics.h.

8.3. Publisher

A publisher is easy to implement. To publish data under the topic “counter” just use counter.publish(data) in any thread.

8.3.1. Program sender.cpp

```
...
#include "topics.h"
```

includes the topics

```
...
    TIME_LOOP(3*SECONDS, 3*SECONDS) {
      PRINTF("Publisher01 sending Counter1 %ld\n", ++cnt);
      counter1.publish(cnt);
    }
...
```

publishes every 3 seconds the incremented counter

8.4. Subscriber

There are many possibilities to implement a receiver of middleware data.

8.4.1. Subscriber put() method, program receiver_simple.cpp

Define a new subscriber by inheriting from “Subscriber”:

```
class SimpleSub :  public Subscriber {
public:
  SimpleSub() : Subscriber(counter1, "simplesub") { }

  long put(const long topicId, const long len, const void* data, ...) {
    PRINTF("SimpleSub - Length: %ld Data: %ld ...
    return 1;
  }
} simpleSub;
```

subscribing for counter1

the put function is called everytime new data has been published; receive the data in this method and send it to a thread via CommBuffer or Fifo (it is also possible to resume a thread when new data is available)

Compile the tutorial receiver_simple.cpp in tutorials/first-steps as described in chapter 2 and execute it. Do not forget to compile it with topics.cpp and sender.cpp:

$ linux-executable topics.cpp sender.cpp receiver_simple.cpp

8.4.2. Subscriber and a CommBuffer, program receiver_combuf.cpp

Define a CommBuffer that is going to be filled by a Subscriber. The thread gets periodically the latest data from the CommBuffer.

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

class ReceiverBuf : public Thread {
  void run () {
    long cnt;
    TIME_LOOP(0, 1.1*SECONDS) {
      buf.get(cnt);
      PRINTF( "ReciverComBuffer - counter1: %ld\n",cnt);
    }
  }
} recbuf;
```

subscriber that filles the CommBuffer with values from topic counter1

the thread gets the latest value

Compile the tutorial receiver_commbuff.cpp in tutorials/first-steps as described in chapter 2 and execute it. Do not forget to compile it with topics.cpp and sender.cpp:

$ linux-executable topics.cpp sender.cpp receiver_commbuff.cpp

To get synchronised data transfer use a SyncFifo like in tutorial receiver_sync.cpp

8.4.3. Putter, program receiver_putter.cpp

Define a new Putter by inheriting from “Putter”:

```
class JustPrint : public Putter {
  bool putGeneric(const long topicId, unsigned int msgLen, ...) {
    PRINTF("%d %ld %ld\n", msgLen, *(long*)msg, topicId);
    return true;
```

is called every time new data is available on defined topics

```
  }
} justPrint;


static Subscriber nameNotImportant01(counter1, justPrint, "justprint01");
static Subscriber nameNotImportant02(counter2, justPrint, "justprint02");
```

subscriber with topic definition – both, counter1 and counter2 will call the putter method of “justprint”

Compile the tutorial receiver_putter.cpp in tutorials/first-steps as described in chapter 2 and execute it. Do not forget to compile it with topics.cpp and sender.cpp:

$ linux-executable topics.cpp sender.cpp receiver_putter.cpp

To receive two counters, implement a sender of the second counter.

8.4.4. Which subscriber is best for what?

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 also good for this. To receive from multiple topics with one method a Putter should be used.

For more information and tutorials about the middleware check out the folders tutorials/middleware and tutorials/alice_bob_charly

9. More Middleware

To see a little more about using the middleware and multicasting, please have a look at the example in the directory gps. Here we have an example of topics with more than one subscribers and of subscribers of more than one topic. 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, and prints the data.

executeit:	shell script to compile and execute the whole example
topics.h:	interface of the topics
topics.cpp:	definition of the topics position and speed
positionsensor.cpp	generates and publishes random postion data
speedcalc.cpp	subscribes position topic and publishes speed data
display.cpp	subscribes position and speed topic and prints the data

Compile all *cpp and see the execution.

Then try to compile without speedcalc.cpp. Do you see the difference?

9.1. program positionsensor.cpp

```
class PositionSensor : public Thread {
```

```
...
        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);
        }
...
```

calculate random movement

publish new position in topic „position”

9.2. program speedcalc.cpp

```
class SpeedCalc :  public Subscriber {
public:
    SpeedCalc() : Subscriber(position, "SpeedCalc") { }
    Pos p0,p1;
    long put(...) {
        p0=p1;
        p1=*(Pos*)data;
        double v = sqrt((p0.x-p1.x)*(p0.x-p1.x)+...);
        speed.publish(v);
        return 1;
    }
} speedCalc;
```

subscribe topic position

calculate and publish speed whenever new position data is published

9.3. program display.cpp

```
static CommBuffer<Pos>    posbuf;
static CommBuffer<double> speedbuf;
static Subscriber namenotimportant1(position, posbuf,   "posreceiverbuf");
static Subscriber namenotimportant2(speed,    speedbuf,
"speedreceiverbuf");

class Display : public Thread {
    void run () {
        TIME_LOOP(1*SECONDS, 1*SECONDS) {
            Pos p;
            double v;
            posbuf.get(p);
            speedbuf.get(v);
            PRINTF( "Position (%3.2f;%3.2f;%3.2f) speed %3.2f\n",.....);
        }
    }
} display;
```

fill buffers with published data

get data from buffers

print data