

Tutorial on the Component Interface

Open RObot COntrol Software

1.2.2

Tutorial on the Component Interface : *Open RObot COnTrol Software* : 1.2.2

Copyright © 2002,2003,2004,2005,2006 Peter SoetensFMTC

Abstract

This document gives an introduction on the different aspects of interfacing an Orocos component. This is an excerpt from the Component Builder's Manual and walks through the 'Hello World' program.

Orocos Version 1.2.2.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation, with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of this license can be found at <http://www.fsf.org/copyleft/fdl.html>.

Table of Contents

| | |
|------------------------|---|
| 1. | 1 |
| 1. Introduction | 1 |
| 2. Hello World ! | 3 |

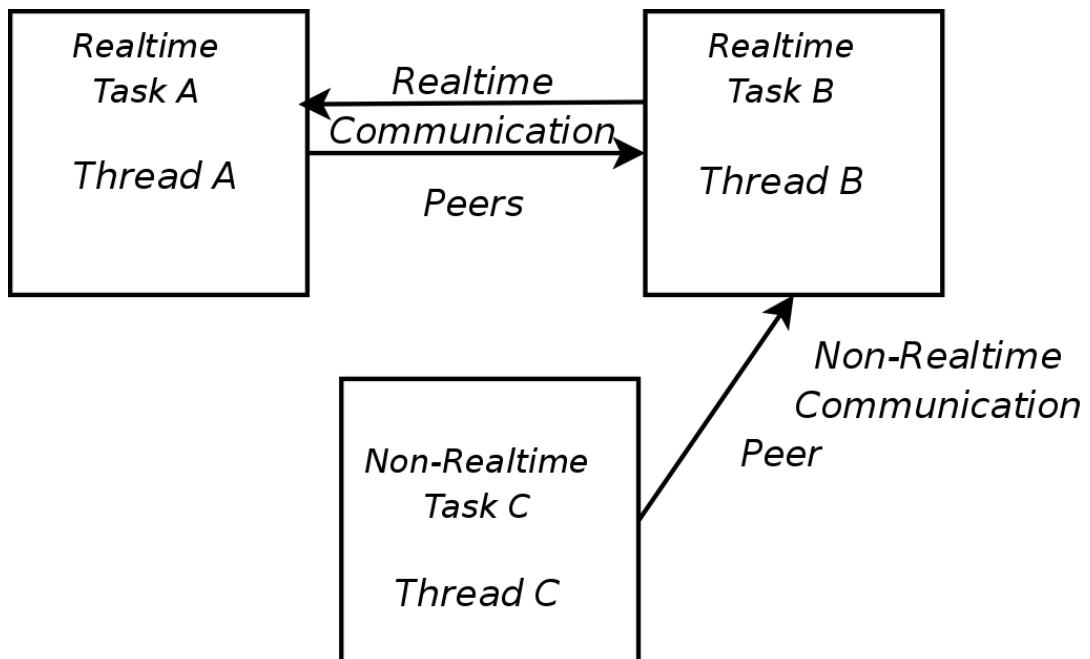
List of Figures

| | |
|--|---|
| 1.1. Tasks Run in Threads | 1 |
| 1.2. Schematic Overview of a TaskContext | 3 |
| 1.3. Schematic Overview of the Hello Component. | 3 |

Chapter 1.

1. Introduction

This manual documents how multi-threaded components can be defined in Orocos such that they form a thread-safe robotics/machine control application. Each control component is defined as a "TaskContext", which defines the "context" in which the component task is executed. The context is built by the five Orocos primitives: Event, Property, Command, Method and Data Port. This document defines how a user can write his own task context and how it can be used in an application.



Tasks run in (periodic) threads.

Figure 1.1. Tasks Run in Threads

A task is a basic unit of functionality which executes one or more (real-time) programs in a single thread. The program can vary from a mere C function over a real-time program script to a real-time hierarchical state machine. The focus is completely on thread-safe time determinism. Meaning, that the system is free of priority-inversions, and all operations are lock-free (also data sharing and other forms of communication such as events and commands). Real-time tasks can communicate with non real-time tasks (and vice verse) transparently.

The Orocos Task Infrastructure enables :

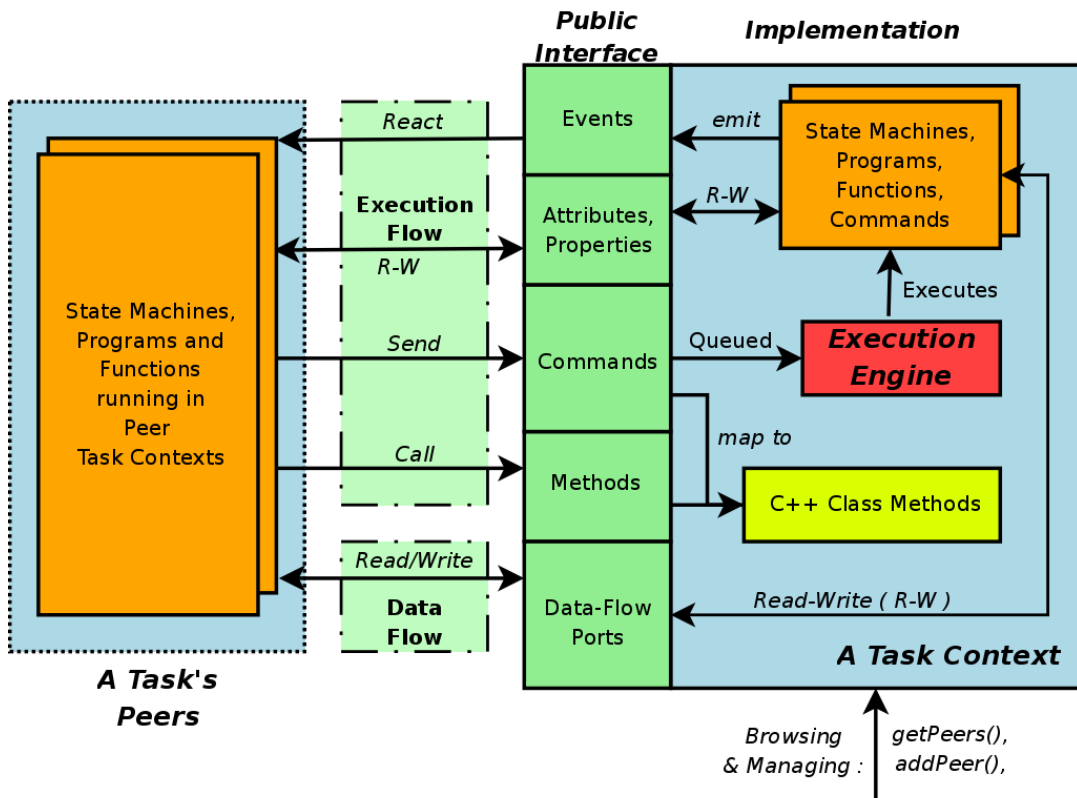
- Lock free, thread-safe, inter-thread function calls.

- Communication between hard Real-Time and non Real-Time threads.
- Deterministic execution time during communication for the higher priority thread.
- Synchronous and asynchronous communication between threads.
- Interfaces for component distribution.
- C++ class implementations for all the above.

This chapter relates to other chapters as such :

| | |
|------------------|--|
| Core Library | provides the Command and Event infrastructure, activity to thread mapping, Properties and lock-free data exchange implementations. |
| Execution Engine | provides a real-time program execution framework. It executes the real-time programs and scripts which interact with other tasks. |
| Orocos Scripting | provides a real-time scripting language which is convertible to a form which can be accepted by the Execution Engine. |

The Scripting chapter gives more details about script syntax for state machines and programs.



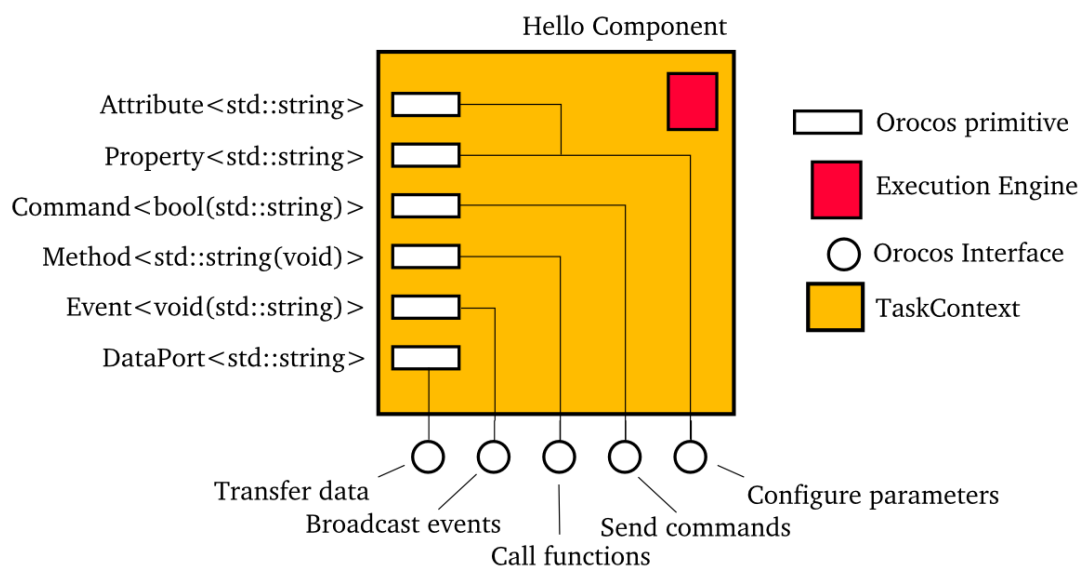
The Execution Flow is formed by Programs and State Machines sending commands, events,... to Peer Tasks. The Data Flow is the propagation of data from one task to another, where one producer can have multiple consumers and vice versa.

Figure 1.2. Schematic Overview of a TaskContext

A task's interface consists of : Attributes and Properties, Commands, Methods, Events and Data Flow ports which are all public. The class TaskContext groups all these interfaces and serves as the basic building block of applications. A component developer 'builds' these interfaces using the instructions found in this manual.

2. Hello World !

This section introduces tasks through the "hello world" application, which can be downloaded from Orococos.org. It contains one TaskContext component which has one instance of each communication primitive.



Our hello world task.

Figure 1.3. Schematic Overview of the Hello Component.

The way we interact with TaskContexts during development of an Orococos application is through the *Task Browser*. The TaskBrowser is a powerful console tool which helps you to explore, execute and debug TaskContexts in running programs. All you have to do is to create a TaskBrowser and call its loop() method. When the program is started from a console, the TaskBrowser takes over user input and output.



Note

The `Orocos::TaskBrowser` is a component of its own which is found in the Orocos Component Library (OCL).

```
#include <ocl/TaskBrowser.hpp>
#include <rtt/os/main.h>
// ...

using namespace Orocos;

int ORO_main( int, char** )
{
    // Create your tasks
    TaskContext* task = ...

    // when all is setup :
    TaskBrowser tbrowser( task );

    tbrowser.loop();
    return 0;
}
```

The `TaskBrowser` uses the GNU readline library to easily enter commands to the tasks in your system. This means you can press `TAB` to complete your commands or press the up arrow to scroll through previous commands.

```
0.016 [ Info  ][main()] ./helloworld manually raises LogLevel to 'Info' (5). See also file
'orocos.log'.
0.017 [ Info  ][main()] **** Creating the 'Hello' component ****
0.018 [ Info  ][ConnectionC] Creating Asyn connection to the_event.
0.018 [ Info  ][ExecutionEngine::setActivity] Hello is periodic.
0.019 [ Info  ][main()] **** Starting the 'Hello' component ****
0.019 [ Info  ][main()] **** Using the 'Hello' component ****
0.019 [ Info  ][main()] **** Reading a Property: ****
0.019 [ Info  ][main()]   the_property = Hello World
0.019 [ Info  ][main()] **** Sending a Command: ****
0.020 [ Info  ][main()]   Sending the_command : 1
0.020 [ Info  ][main()] **** Calling a Method: ****
0.020 [ Info  ][main()]   Calling the_Method : Hello World
0.020 [ Info  ][main()] **** Emitting an Event: ****
0.021 [ Info  ][main()] **** Starting the TaskBrowser ****
0.021 [ Info  ][TaskBrowser] Creating a BufferConnection from the_buffer_port to
the_buffer_port with size 13
0.021 [ Info  ][TaskBrowser] Connected Port the_buffer_port to peer Task Hello.
0.022 [ Info  ][Hello] Creating a DataConnection from the_data_port to the_data_port
0.022 [ Info  ][Hello] Connected Port the_data_port to peer Task TaskBrowser.
Switched to : Hello
0.023 [ Info  ][main()] Entering Task Hello
0.023 [ Info  ][Hello] Hello Command: World
0.023 [ Info  ][Hello] Receiving Event: Hello World
```

```
This console reader allows you to browse and manipulate TaskContexts.
You can type in a command, event, method, expression or change variables.
(type 'help' for instructions)
TAB completion and HISTORY is available ('bash' like)
```

```
In Task Hello. (Status of last Command : none )
(type 'ls' for context info) :
```

The first [**Info**] lines are printed by the Orocos Logger, which has been configured to display informative messages to console. Normally, only warnings or worse are displayed by Orocos. You can always watch the log file 'orocos.log' in the same directory to see all messages. After the [**Log Level**], the [**Origin**] of the message is printed, and finally the message itself. These messages leave a trace of what was going on in the main() function before the prompt appeared.

Depending on what you type, the TaskBrowser will act differently. The built-in commands **cd**, **help**, **quit** and **ls** are seen as commands to the TaskBrowser itself, if you typed something else, it tries to evaluate your command to an expression and will print the result to the console. If you did not type an expression, it tries to parse it as a command to a (peer) task. If that also fails, it means you made a typo and it prints the syntax error to console.

```
In Task Hello. (Status of last Command : none )
(type 'ls' for context info) :1+1
    Got :1+1
    = 2
```

To display the contents of the current task, type **ls**, and switch to one of the listed peers with **cd**, while **cd ..** takes you one peer back in history. Since there are no peers other than the TaskBrowser itself, one can not **cd** anywhere in this example.

```
In Task Hello. (Status of last Command : none )
(type 'ls' for context info) :ls

Listing Hello :

Attributes :
(Attribute)  string the_attribute    = Hello World
(Attribute)  string the_constant     = Hello World
(Property)   string the_property    = Hello World      (Hello World Description)

Interface :
Methods     : the_method isRunning start stop trigger update
Commands    : the_command
Events      : the_event

Objects     :
            this ( )
```

```
the_data_port ( A Task Object. )
the_buffer_port ( A Task Object. )

Ports      : the_data_port the_bufferPort
Peers      : TaskBrowser
```



Note

To get a quick overview of the commands, type **help**.

First you get a list of the Properties and Attributes (alphabetical) of the current component. Properties are meant for configuration and can be written to disk. Attributes are solely for run-time values. Each of them can be changed (except constants.)

Next, the interface of this component is listed: One method is present *the_method*, one command *the_command* and one event *the_event*. They all print a 'Hello World' string when invoked.

In the example, the current task has only three objects: *this*, *the_data_port* and *the_buffer_port*. The *this* object serves as the public interface of the Hello component. These objects contain methods, commands or events. The next two objects are created to represent the data ports of the Hello component and contain the operations to send or receive data or query connection status.

Last, the peers are shown, that is, the components which are connected to this component. The HelloWorld component is a stand-alone component and has only the TaskBrowser as a peer.

To get a list of the Task's interface, you can always type an object name, for example *this*.

```
In Task Hello. (Status of last Command : none )
(type 'ls' for context info) : this
  Got :this

Printing Interface of 'Hello' :

Command  : bool the_command( string the_arg )
  Hello Command Description
  the_arg : Use 'World' as argument to make the command succeed.
Method   : string the_method( )
  Hello Method Description
Event    : void the_Event( string the_data )
  Hello Event Description
  the_data : The data of this event.
```

Now we get more details about the commands, methods and events registered in the public interface. We see now that the *the_command* command takes one argument

as a string, or that the *the_method* method returns a string. One can invoke each one of them:

```
In Task Hello. (Status of last Command : none )
(type 'ls' for context info) :the_method()
  Got :the_method()
= Hello World
```

Methods are called directly and the TaskBrowser prints the result. The return value of the *the_method()* was a string, which is "Hello World".

When a command is entered, it is sent to the Hello component, which will execute it on behalf of the sender. The different stages of its lifetime are displayed by the prompt. Hitting enter will refresh the status line:

```
In Task Hello. (Status of last Command : none )
(type 'ls' for context info) :the_command("World")
  Got :the_command("World")

In Task Hello. (Status of last Command : queued)
1021.835 [ Info ][Hello] Hello Command: World
(type 'ls' for context info) :

1259.900 [ Info ][main()] Checking Command: World
In Task Hello. (Status of last Command : done )
(type 'ls' for context info) :
```

A Command might be rejected (return false) in case it received invalid arguments:

```
In Task Hello. (Status of last Command : done )
(type 'ls' for context info) :the_command("Belgium")
  Got :the_command("Belgium")

In Task Hello. (Status of last Command : queued )
(type 'ls' for context info) :
1364.505 [ Info ][Hello] Hello Command: Belgium

In Task Hello. (Status of last Command : fail )
(type 'ls' for context info) :
```

Besides sending commands to tasks, you can alter the attributes of any task, program or state machine. The TaskBrowser will confirm validity of the assignment with 'true' or 'false' :

```
In Task Hello. (Status of last Command : none )
(type 'ls' for context info) :the_attribute
```

```

    Got :the_attribute
= Hello World
In Task Hello. (Status of last Command : none )
(type 'ls' for context info) :the_attribute = "Veni Vidi Vici !"
    Got :the_attribute = "Veni Vidi Vici !"
= true

In Task Hello. (Status of last Command : none )
(type 'ls' for context info) :the_attribute
    Got :the_attribute
= Veni Vidi Vici !

```

Finally, let's emit an Event. The Hello World Event requires a payload. A callback handler was registered by the component, thus when we emit it, it can react to it:

```

In Task Hello. (Status of last Command : none )
(type 'ls' for context info) :the_event(the_attribute)
    Got :the_event(the_attribute)
= true
In Task Hello. (Status of last Command : none )
(type 'ls' for context info) :
354.592 [ Info ] [Hello] Receiving Event: Veni Vidi Vici !

```

The example above passed the `the_attribute` object as an argument to the event, and it was received by our task correctly. Events are related to commands, but allow broadcasting of data, while a command has a designated receiver.

The Data Ports can be accessed through the `the_data_port` and `the_buffer_port` object interfaces. Once again, we can inspect the interface of an object by typing its name:

```

In Task Hello. (Status of last Command : none )
(type 'ls' for context info) :the_data_port
    Got :the_data_port

```

Printing Interface of 'the_data_port' :

```

Method   : string Get()
          Get the current value of this Data Port
Method   : void Set( string const& Value )
          Set the current value of this Data Port
          Value : The new value.

```

The `the_data_port` object has two methods: `Get()` and `Set()`. Since data ports are used for sending unbuffered data packets between components, this makes sense. One can interact with the ports as such:

```

In Task Hello. (Status of last Command : none )

```

```
(type 'ls' for context info) :the_data_port.Set("World")
  Got :the_data_port.Set("World")
= (void)
```

```
In Task Hello. (Status of last Command : none )
(type 'ls' for context info) :the_data_port.Get()
  Got :the_data_port.Get()
= World
```

When a value is Set(), it is sent to whatever is connected to that port, when we read the port using Get(), we see that the previously set value is present. The advantage of using ports is that they are completely thread-safe for reading and writing, without requiring user code. The Hello component also contains a the_buffer_port for buffered data transfer. You are encouraged to play with that port as well.

Remember that the TaskBrowser was a component as well ? When a user enters **ls**, the interface of the visited component is listed. It is also possible to get an 'outside' view of the visited component, through the eyes of an external component. The **leave** allows a view from within the TaskBrowser itself:

```
In Task Hello. (Status of last Command : none )
(type 'ls' for context info) :leave
1001.607 [ Info  ][main()] Watching Task Hello

Watching Task Hello. (Status of last Command : none )
(type 'ls' for context info) :ls

Listing TaskBrowser :

Attributes  : (none)

Interface  :
Methods    : isRunning start stop trigger update
Commands   : (none)
Events     : (none)

Objects    :
  this ( )
  the_data_port ( A Task Object. )
  the_buffer_port ( A Task Object. )

Ports      : the_data_port the_buffer_port
Hello Peers : TaskBrowser
```

The following things are noteworthy: 'ls' shows now the contents of the TaskBrowser itself and no longer of the Hello Component. The TaskBrowser has the same ports as the component it visits: *the_data_port* and *the_buffer_port*. These were created at run-time and allow to write or read ports from the visited component.

One can enter the 'inside' view again by entering **enter**:

```
Watching Task Hello. (Status of last Command : none )
(type 'ls' for context info) :enter
1322.653 [ Info ][main()] Entering Task Hello

In Task Hello. (Status of last Command : none )
(type 'ls' for context info) :
```

Last but not least, hitting TAB twice, will show you a list of possible completions, such as peers or commands :

```
In Task Hello. (Status of last Command : none )
(type 'ls' for context info) :
the_attribute  the_event  cd ..  quit
the_buffer_port. the_method  help  this.
the_command  the_property  leave
the_constant  TaskBrowser. list
the_data_port.  cd  ls
(type 'ls' for context info) :
```

TAB completion works even across peers, such that you can type a TAB completed command to another peer than the current peer.

In order to quit the TaskBrowser, enter **quit**:

```
In Task Hello. (Status of last Command : none )
(type 'ls' for context info) :quit

1575.720 [ Info ][ExecutionEngine::setActivity] Hello is disconnected from its activity.
1575.741 [ Info ][Logger] Orocos Logging Deactivated.
```

The TaskBrowser Component is application independent, so that your end user-application might need a more suitable interface. However, for testing and inspecting what is happening inside your real-time programs, it is a very useful tool. The next sections show how you can add properties, commands, methods etc to a TaskContext.



Note

If you want a more in-depth tutorial, see the 'task-intro' example for a TaskBrowser which visits a network of three TaskContexts.