WP9 – Validation on platform

Deliverable D9rb.4_rep
Robotic Application V2 Development Report

WP9 – Validation on platform : Deliverable D9rb.4_rep – Robotic Application V2 Development Report

by F. RUSSOTTO (CEA)

Published: january 2005

Table of Contents

Document Presentation

Project Coordinator

Organization:	UPVLC
Responsible person:	Alfons Crespo
Address:	Camino Vera, 14, 46022 Valencia, Spain
Phone:	+34 963877576
Fax:	+34 963877576
Email:	alfons@disca.upv.es

Participant List

Role	Id.	Participant Name	Acronym	Country
CO	1	Universidad Politecnica de Valencia	UPVLC	E
CR	2	Scuola Superiore Santa Anna	SSSA	I
CR	3	Czech Technical University in Prague	CTU	CZ
CR	4	Commissariat a l'Energie Atomique	CEA	FR
CR	5	Unicontrols	UC	CZ
CR	6	MNIS	MNIS	FR
CR	7	Visual Tools S.A.	VT	E

Document version

Release	Date	Reason of change
1_0	24/01/2005	First release

Chapter 1. Introduction

1.1 Summary

This document is the development report of the Robotic application, final release (v2). The Robotic application is intended to demonstrate the efficiency of Linux, of RT-Linux, and of the real-time system components developed as part of the OCERA project (WP4-7).

This document is not intended to provide a full and detailed description of the whole application components for two main reasons:

1) The OCERA robotic application is a porting to Linux/RTLinux of an existing industrial robotic application developed and under license of HAPTION SA (www.haption.com). HAPTION SA agreed that the robotic application be used as a validation and benchmark tool for OCERA, provided no strategic parts of the code are detailed in any public document.
2) The application has to be considered as a validation tool within the OCERA project. Describing the whole application would require a huge document in order to describe each algorithm implemented. This does not make sense from the OCERA component users point of view. This way, only the general application overview and application components that interface with Linux, RT-Linux and the OCERA system components will be described in detail in this document.

1.2 Description

The OCERA robotic application consists of the servo-control software component of a master robot arm (called Virtuose), which is designed for haptic purposes by HAPTION SA[1]. The Virtuose robot arm is a man/machine servo-mechanical interface allowing users to handle objects remotely with high (say tactile) precision and realism. For the OCERA robotic application, the Virtuose robot arm will be used in a 3 dimensional (3D) virtual environment (commonly called "virtual theater"), consisting of a large screen, which displays a graphical virtual scene made of 3D objects, as shown below. The Virtuose robot arm allows users to handle virtual objects of the 3D scene with realistic 6 DoF force feedback so that users can feel weight, inertia, and collisions between objects, as if they were handling real objects in a real environment.

[1] Haptics is related to tactile robotics.

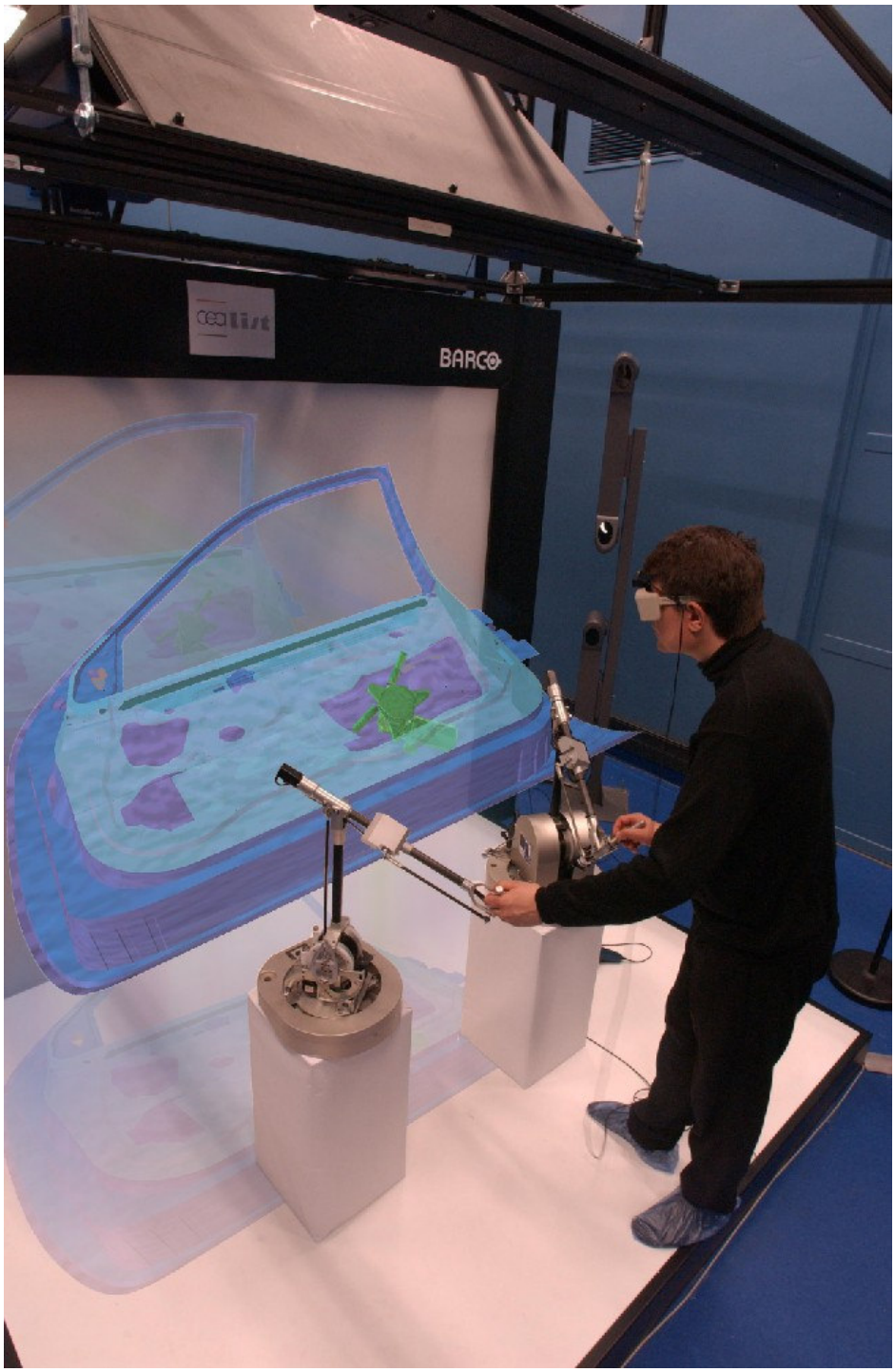


Figure 1 : A couple of Virtuose robot arms at work

The above figure shows a subjective view of the demonstration that will be carried out for the OCERA project. This demo has been developed for a French car manufacturer whose objective is to implement new real-time tools for mounting feasibility analysis. The goal of the demo is to show the feasibility of mounting a window winder into a car door in a limited time. This information is useful to car manufacturers to lay out mounting times of car components at the very beginning stage of conception, and to optionally modify the design of components whose mounting time would not fit requirements.

The application implementation requires two pieces of software to run: a controller application and a client application, each running on a dedicated computer[2], and connected together by Ethernet. The client software runs under the MS Windows operating system, while the controller software runs under the RTLinux/OCERA real-time kernel.

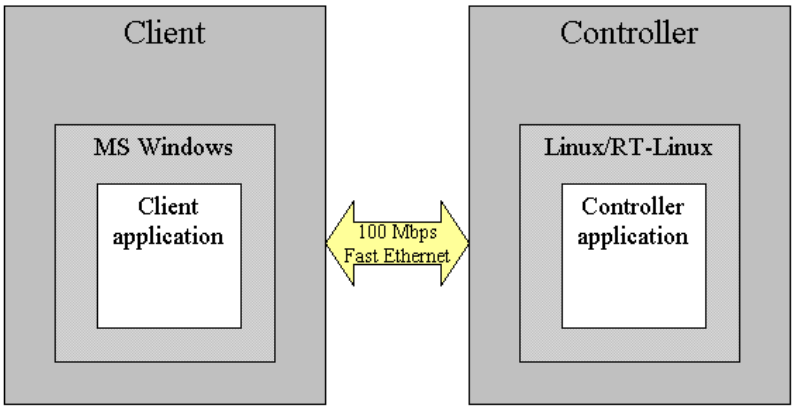


Figure 2: Application implementation

The description of each application follows. OCERA is mainly concerned by the controller application, running the robotic control (and hard real-time) part of the overall haptic application. The controller application requires high performance from the real-time kernel on top of which it runs for two main reasons. First, to achieve the expected mechanical stiffness and precision that impart the tactile realism. Second, to guarantee the robotic control stability required to avoid user injury.

1.2.1 The Client application

The client application is a Win32 executable running on a Windows desktop computer. It is connected to the controller application through an Ethernet connection, using UDP/IP protocol. Data transmitted is encoded/decoded to/from the Sun® XDR® encoding format to avoid data corruption due to communication between different platforms.

In OCERA application v1, the client application was based on a simplified physics/dynamics engine and a simplified visual/graphics rendering engine (see D9rb.3_rep). In application v2, "true" high-end industrial graphics and physics engines are used instead: These are the Virtools graphics rendering engine from Virtools (http://www.virtools.com) and the Vortex physics/dynamics engine from CMLabs (http://www.cm-labs.com/products/vortex).

The client application runs a virtual environment consisting of a car door and a window winder (3D numeric models including both graphical and physical characteristics). The client application computes graphics rendering (visual aspects) and physics rendering (i.e. dynamics aspects such as weights, inertias, stiffnesses and collisions between virtual objects).

When connected to the Client, the controller application can retrieve the movement of any virtual object simulated within the Client and can "virtually" apply forces and torques to it, in order to modify its movement. This allows us to link mechanically (but virtually) the Virtuose robotic arm handle to any object of the virtual environment, as described below.

[2] The client application may actually be distributed between more than one computer; this will not be described here however.

1.2.2 The Controller application

The controller application is a full kernel space implementation written in standard C, running under RTLinux/OCERA-1.1 on a x86 embedded Controller. It is mostly hard real-time, except for the soft real-time communication with the Client, which is processed asynchronously (but also in kernel space, in order to optimize soft real-time performance). It is started by loading the application kernel module into the Controller Linux kernel.

The controller application performs three main tasks:
- Robotic control in force and torque of the Virtuose robot arm handle, in 6D Cartesian space.
- Computation of force and torque to be applied to both the Virtuose robot arm handle, and to a particular virtual object, in order to implement a virtual mechanical link between them. Such mechanical link has typically the characteristics of a spring/absorber, and this computation is commonly called bilateral force/torque coupling.
- Dynamics data exchange with the Client (forces and torques from Controller to Client, positions and speeds from Client to Controller).

Bilateral coupling allows fully symmetrical behaviors between the Virtuose handle and the virtual object it is coupled with. As a result of the coupling computation, we have the following:
- Any force or torque applied to the Virtuose handle will be applied to the virtual object at the same time and vice-versa,
- Any movement of the Virtuose handle will lead to a homothetic movement of the virtual object and vice-versa.

The application includes several components. Most of these components will not be described in details in this document because they are robotics or haptics related, and they do not interact with the real-time kernel. They are briefly described in this section to provide the reader with some information about how the application works.

- System interface components: These components provide a common platform interface to other application components, in order to ease application portability toward several real-time operating systems or kernels. They can be used within the context of any real-time application in order to improve the application portability. Most of these components strongly interface with Linux, RT-Linux and OCERA components. They are fully described in the following chapters.
- Robotic & haptic components: theses components provide common robotic and haptic features to the application. These components use the system-interface components but do not have any other interface with the OS, and then will not be detailed.
- Application component: this component is the Controller application top level component. The main functions of this component are described in the following chapters.

1.2.2.1 The system interface components

The system interface components have been strongly reduced in application version 2, compared to version 1 (see D9rb3). First, we have benefited from the new features of RTLinux. Another reason for such reduction was the need to consolidate the application, following on the deep redesign of the application/system interface.

Figure 3 shows the dependency tree of the generic real-time component, **rt_interface**. Yellow components are RTLinux/OCERA kernel components. Pink ones are OCERA new components, and the white one is **rt_interface**. The **rt_interface** component mostly encapsulates used RT kernel primitives such as task management and ITCs, in order to improve application portability. It is fully detailed in Chapter 2.

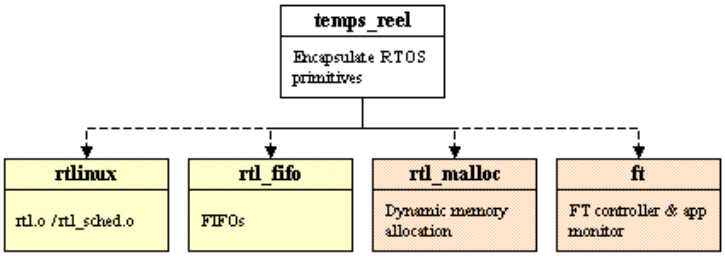


Figure 3: The rt_interface system component dependency tree

In order to improve application portability, a UNIX compatibility header set has been developed. The structure of this header set is very similar to the standard UNIX one. Each header file of the set includes a bunch of required Linux and RTLinux/OCERA kernels, and OCERA components header files. It defines some useful types, macros and constants that are compliant (though very partially) with the SUS[3]. This header set strongly eases the application's portability from standard UNIX to RTLinux/OCERA by providing the application with a UNIX-like API. Figure 4 shows the include tree of the UNIX compatibility headers set. Brown headers are standard UNIX (/usr/include) or Linux kernel (/usr/include/linux) headers, yellow headers are RTLinux/OCERA headers, and white headers are part of the compatibility set. This header set is fully described in Chapter 3.

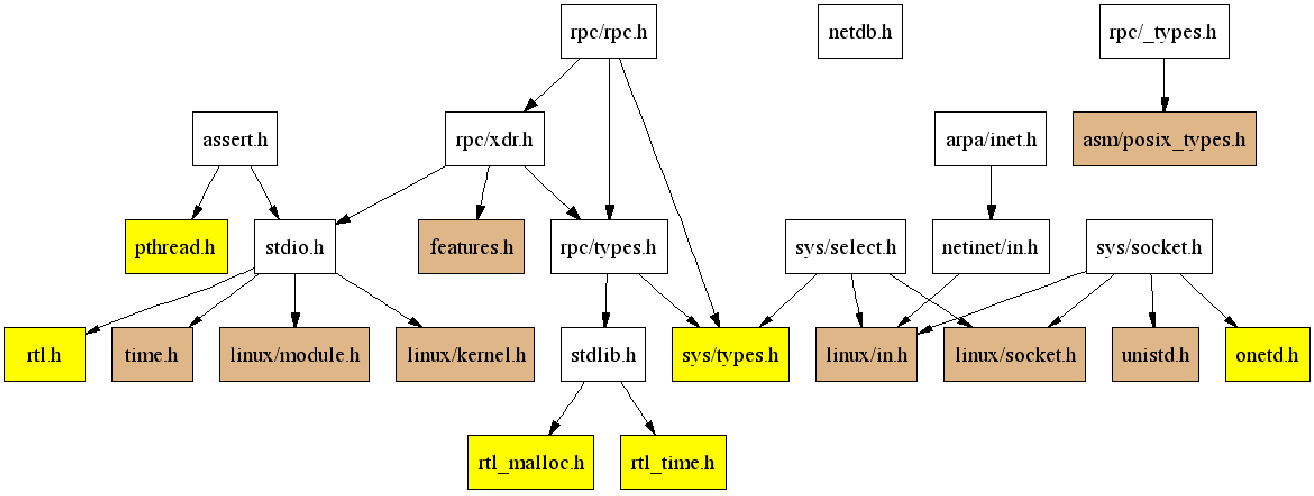


Figure 4: Unix compatibility headers set

[3] Single UNIX Specification

1.2.2.2 Robotics, haptics, and application-specific components

The following figure shows the simplified dependency tree of the application kernel-module. Components represented in yellow are RTLinux/OCERA kernel components, orange ones are OCERA components, and white ones are application components.

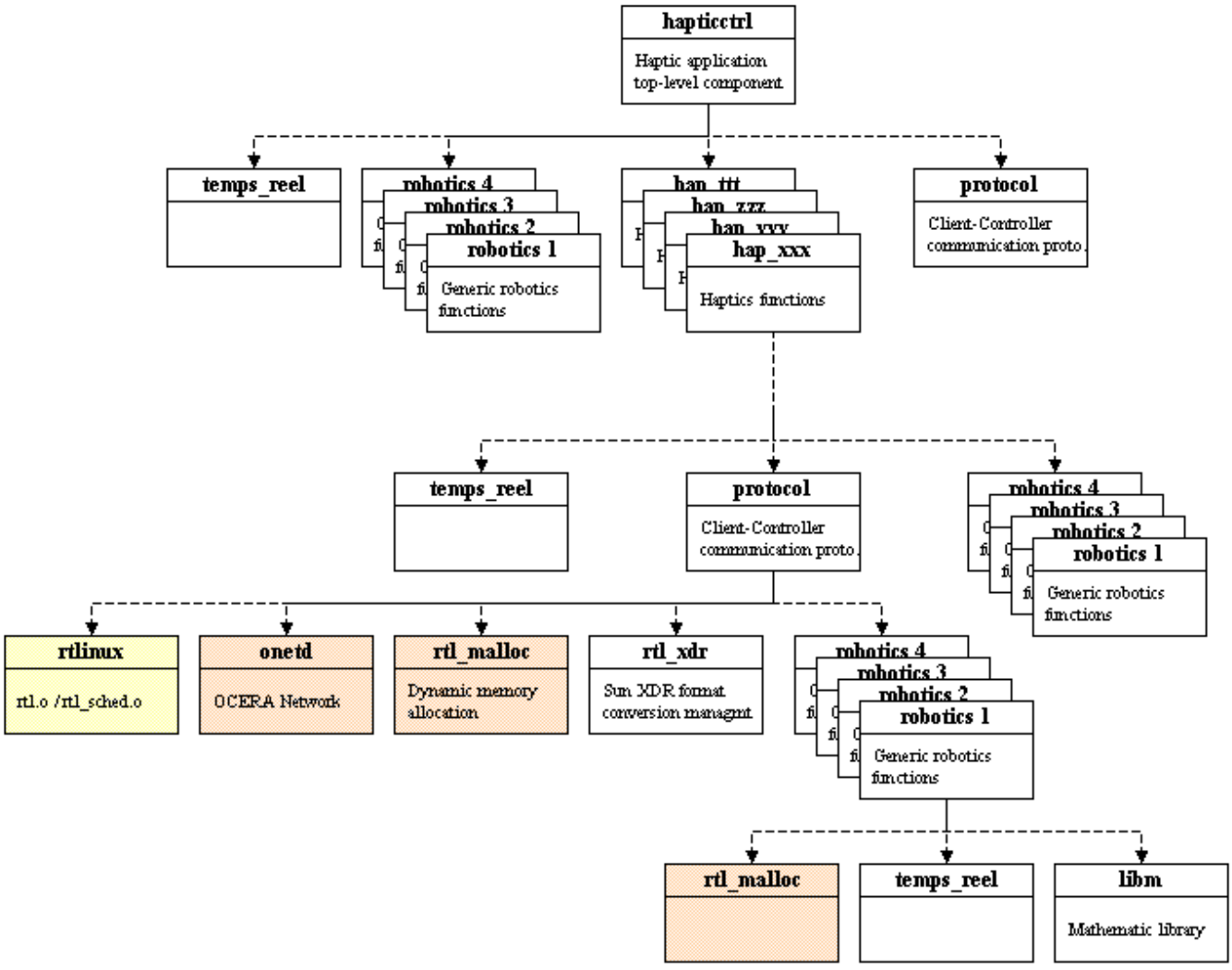


Figure 5: Controller application dependency tree overview

Here is a brief description of the components:

- robotics #: these components provide common robotics features; they are not detailed in the following sections.
- hap_###: these components provide common haptics features; they are not detailed in the following sections.
- protocol: this component provides a simple communication protocol for communication between the controller application and the client application. This component is not described anymore in the documentation because it now uses the OCERA networking component ONetD, which provides to the component the standard BSD socket interface it needs. Compared to previous versions of the protocol component (robotic application v1, see D9rb.3_rep), this component did not need any specific modification in order to work on top of RTLinux/OCERA and ONetD component.

- **hapticctrl**: this component is the kernel-module top-level component; it includes all application specific definitions and the main servo-control task code. This component is briefly described in Chapter 3.

1.2.2.3 Tasks view

The following figure shows a tasks view of the controller application. Compared to application v1 (see D9rb.3), all tasks now run in Linux kernel space, the CLOCK task is no more used (saving CPU time), and the INIT and PROTO_IN tasks are fault tolerant tasks (using OCERA fault-tolerance controller and application monitor components).

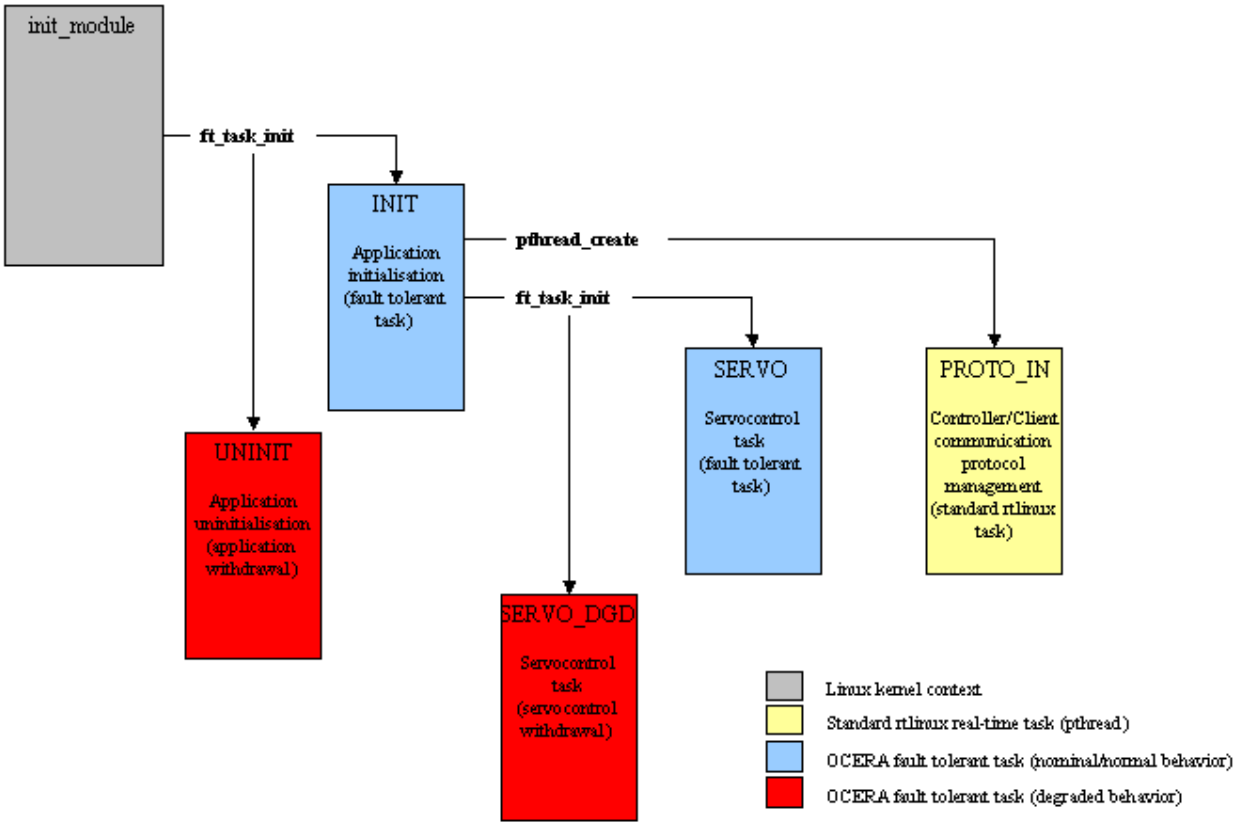


Figure 6: Controller application tasks view

The tasks perform the following jobs:

- **init_module** (Linux kernel context): application startup task
  - starts the application initialization task as a OCERA fault tolerant task (INIT/UNINIT pair).
- **INIT** (real-time fault tolerant task): application initialisation task
  - allocates and initializes all application data structures.
  - starts the controller/client communication protocol management task PROTO_IN as a standard RTLinux thread (pthread_create).
  - starts the servo-control task as an OCERA fault tolerant task (SERVO/SERVO_DGD pair).
- **UNINIT** (degraded mode of the INIT task): application fallback task (this task is switched up by OCERA fault-tolerant controller if the INIT task fails).
  - depending on the initialization stage reached in the INIT task, kills all created tasks and frees all previously allocated application data structures.

- **PROTO_IN** (standard RTLinux real-time thread): Controller ← Client communication management.
    - receives incoming data from the client application (using socket services provided by OCERA network component: ONetD).
    - decodes and stores data in a shared buffer for use by the servo-control task (SERVO).
- **SERVO** (real-time fault tolerant task): periodic servo-control task (1 ms period).
    - samples Virtuose robot-arm position and speed from the input boards.
    - computes robotic arm / virtual object coupling using client application data received by PROTO_IN.
    - computes robot-arm servo-control commands.
    - send computed commands to the robotic arm output boards.
    - send back coupling data to the client application (using ONetD socket).
- **SERVO_DGD** (degraded mode of the SERVO task): servo-control fallback (this task is switched up by OCERA fault-tolerant controller if the SERVO task fails).
    - applies adequate commands to the output cards so that the robot-arm is locked up at its current position.

Figure 7 shows a typical chronogram of the active tasks at runtime: SERVO and PROTO_IN. The figure does not show fallback task SERVO_DGD as the SERVO task did not fail here. The figure was generated by kiwi from a log file generated using the OCERA POSIX trace utility.

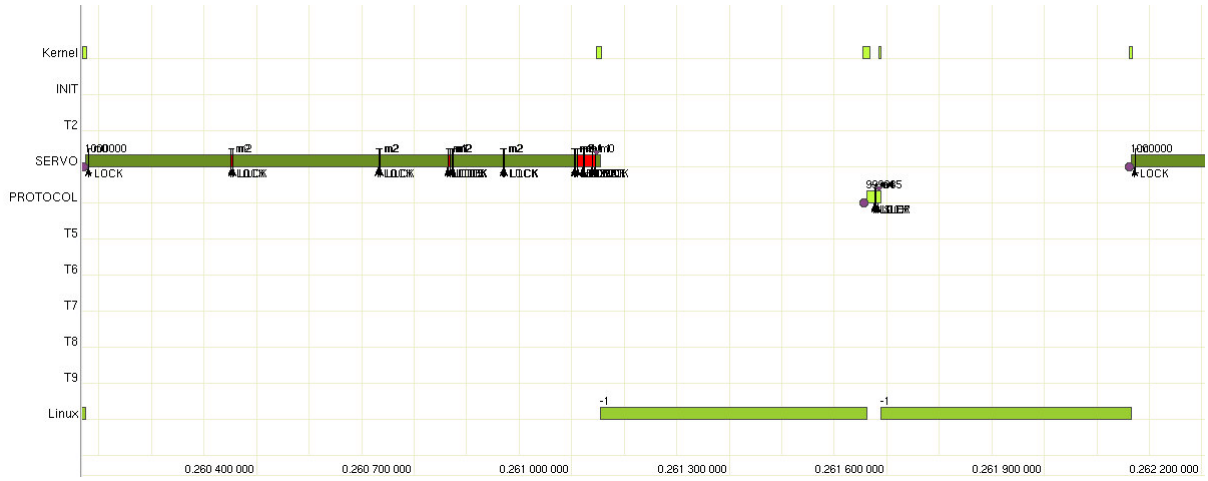


Figure 7: Chronogram of typical tasks at runtime

On the figure, one can clearly see the arrival of an incoming IP packet during the Linux CPU time. This induces pre-emption of the Linux kernel by the PROTO_IN task in order to handle the packet.

Chapter 2. rt_interface component

2.1 Summary

Name:

rt_interface (stands for: real-time system/kernel interface)

Description:

Encapsulates common RT system/kernel primitives for application portability improvement

Author:

F. Russotto (russotto.at.cea.fr)

Reviewer:

J. Brisset (julien.brisset.at.cea.fr)

Layer:

Low level application layer

Version:

2.0

Status:

Stable

Dependencies:

rtl, rtl_sched, rtl_fifo, rtl_malloc, ftappmonitor, ftcontroller.

Release date:

MS 5

2.2 Description

This component encapsulates some common primitives of the RT kernel, such as scheduling, synchronization, MUTEX and timer primitives. The component is designed so that implementation for other RT kernels or operating systems (RTAI, VxWorks, ...) is made possible using #ifdef / #endif blocks. This ensures application code portability toward several platforms.

2.3 API / compatibility

`RT_sem_synchro_create`
Create a synchronization semaphore

`RT_sem_mutex_create`
Create a MUTEX semaphore

RT_sem_destroy
: Destroy a synchronization or MUTEX semaphore

RT_sem_check
: Try lock / try wait a semaphore (non-blocking)

RT_sem_wait
: Lock / wait a semaphore (blocking)

RT_sem_post
: Unlock / post a semaphore

RT_task_create
: Create a Real-Time task / thread

RT_task_destroy
: Destroy (cancel) a RT task

RT_ptask_create
: Create a Real-Time periodic task / thread

RT_ptask_destroy
: Destroy (cancel) a RT task

RT_sleep
: Suspend the current task for a given delay

RT_fifo_init
: Initialize fifo communication channels

RT_fifo_uninit
: Uninitialize fifo communication channels

RT_fifo_open
: Create a communication channel based on a fifo

RT_fifo_open_fd
: Create a communication channel from a fifo number

RT_fifo_fd
: Return fifo number associated to a communication channel

RT_fifo_read
: Read data from a communication channel (blocking)

RT_fifo_write
: Write data to a communication channel

RT_fifo_close
: Close a communication channel

RT_sem_synchro_create

Synopsis

```
RtSem RT_sem_synchro_create(void)
```

Include

"tr.h"

Return value

ID of the newly created semaphore or NULL if an error occurred.

Description

This function allocates needed memory and creates a synchronization semaphore using the sem_init() function of the RT-Linux API.

WARNING: a unique semaphore should not be used to synchronize more than one task. To synchronize multiple tasks, use a barrier instead.

See also

RT_sem_mutex_create, RT_sem_destroy, RT_sem_wait, RT_sem_post

---

RT_sem_mutex_create

Synopsis

```
RtSem RT_sem_mutex_create(void)
```

Include

"tr.h"

Return value

ID of the newly created mutex or NULL if an error occurred

Description

This function allocates needed memory and creates a mutual exclusion semaphore that can be used to protect concurrent accesses to a shared resource. It uses standard pthread_mutex_init() of the RT-Linux API to do so.

See also

RT_sem_synchro_create, RT_sem_destroy, RT_sem_wait, RT_sem_post

---

RT_sem_destroy

Synopsis

```
int RT_sem_destroy(RtSem sem)
```

Include

"tr.h"

Parameters

sem

semaphore to be deleted

Return value

0 if successful, -1 otherwise.

Description

This function destroys a previously created synchronization or mutex semaphore. It uses either the sem_destroy() or the pthread_mutex_destroy() of the RT-Linux API, depending on semaphore type.

See also

RT_sem_synchro_create, RT_sem_mutex_create

RT_sem_check

Synopsis

```
int RT_sem_check(RtSem sem)
```

Include

"tr.h"

Parameters

sem

semaphore to be locked/waited

Return value

0 if sem has been succesfully locked/waited, -1 otherwise

Description

This function performs a non-blocking lock/wait on the given semaphore. This uses either a sem_try_wait() or a pthread_mutext_trylock() call depending on the semaphore type. If sem is being used by another task (locked or waited), the function returns -1 immediately without blocking. If sem is not in use by another task, the function returns 0, and sem becomes locked/waited.

See also

RT_sem_synchro_create, RT_sem_mutex_create

RT_sem_wait

Synopsis

```
int RT_sem_wait(RtSem sem)
```

Include

"tr.h"

Parameters

sem

semaphore to be taken

Return value

0 if successful, -1 if an error occurred

Description

This function locks / waits the given semaphore. It uses either the sem_wait() or the pthread_mutex_lock() functions, depending on the semaphore type. If sem is being used by another task (locked or waited), the function blocks indefinitely while waiting for semaphore unlocking/posting (calling task is suspended). As soon as sem is unlocked/posted (or if sem was not in use), RT_sem_wait() returns 0, sem becomes locked/waited and the calling task is woken up if possible. RT_sem_wait() can be used either to grant access to a shared resource or to suspend the current task for synchronization by another task.

See also

RT_sem_synchro_create, RT_sem_mutex_create

RT_sem_post

Synopsis

```
int RT_sem_post(RtSem sem)
```

Include

"tr.h"

Parameters

sem

semaphore to be unlocked / posted

Return value

0 if successful, -1 otherwise

Description

This functions unlocks / posts the given semaphore. This uses either pthread_mutex_lock() or sem_post() depending on sem type. This function can be used either to synchronize a pending task or to give access back to a shared resource.

See also

RT_sem_synchro_create, RT_sem_mutex_create

RT_task_create

Synopsis

```
RtTask RT_task_create(const char *name, unsigned int prio, size_t
                      stacksize, (void *)(*func)(void *), void *arg)
```

Include

"tr.h"

Parameters

name

name of the task to be created (human readable format)

prio

priority of the task (0 is the highest)

stacksize

task stack size

func

pointer to the task entry point function

arg

arguments to be passed to the task entry point function

Return value

ID of the newly created task or NULL if an error occurred

Description

This function creates a Real-Time task (or thread). Parameter name can be given to identify the task, though this parameter is not used in RT-Linux function calls. Parameter prio should be given according to the following standard: 0 is the highest priority and sched_get_priority_max(SCHED_FIFO) is the lowest one.

RT_task_create() performs the following steps:

- initialize pthread attributes using parameters func and arg
- set pthread scheduling attributes using priority = sched_get_priority_max(SCHED_FIFO) - prio
- dynamically allocates memory for task stack using the malloc() function provided by the rtl_malloc OCERA component (this allows creating a RT task dynamicaly within the context of another running RT task) and parameter stacksize
- set de-tasked mode for the created task
- enable floating point operations for the created task
- create the RT task using standard pthread_create()

See also

RT_ptask_create, RT_task_destroy

RT_task_destroy

Synopsis

```
int RT_task_destroy(RtTask task)
```

Include

"tr.h"

Parameters

task

task to be deleted (cancelled)

Return value

0 if successful, -1 otherwise

Description

This function destroys (i.e. cancels and joins) the specified Real-Time task, and frees allocated memory resource.

See also

RT_task_create,

RT_ptask_create

Synopsis

```
RtTask RT_ptask_create(const char *name, unsigned int prio, size_t
                       stacksize, double period, double start_time, (void
                       *)(*func)(void *), void *arg)
```

Include

"tr.h"

Parameters

name

name of the task to be created (in human readable form)

prio

priority of the task (0 is the highest)

stacksize

task stack size

period

task period in seconds

start_time

task start time (system clock reference)

func

pointer to the task entry point function

arg

arguments to be passed to the task entry point function

Return value

ID of the newly created task or NULL if an error occurred

Description

- This function creates a Real-Time task (or thread) the same way as RT_task_create then makes it periodic.

See also

RT_task_create, RT_ptask_destroy

RT_ptask_destroy

Synopsis

```
int RT_ptask_destroy(RtTask task)
```

Include

"tr.h"

Parameters

task

task to be deleted (cancelled)

Return value

0 if successful, -1 otherwise

Description

This function destroys (i.e. cancels and joins) the specified Real-Time periodic task and frees allocated memory resource.

See also

RT_ptask_create,

RT_fttask_create

Synopsis

```
RtFtTask      RT_fttask_create(const      char      *name,      RtFtTaskParams
                       normal_task_params,                     RtFtTaskParams
                       degraded_task_params)
```

Include

"tr.h"

Parameters

name

name of the FT task to be created (human readable)

normal_task_params

parameters set for the normal mode task

degraded_task_params

parameters set for the degraded mode task

Return value

ID of the newly created task or NULL if an error occurred

Description

This function creates a Fault-Tolerant Real-Time task. This service is based on the OCERA Fault-Tolerant controller and application monitor components.

See also

RT_task_create, RT_fttask_destroy

RT_fttask_destroy

Synopsis

```
int RT_fttask_destroy(RtFtTask task)
```

Include

"tr.h"

Parameters

task

ft task to be deleted (i.e. cancelled)

Return value

0 if successful, -1 otherwise

Description

This function destroys a Real-Time Fault-Tolerant periodic task and frees allocated memory resource.

See also

RT_fttask_create,

RT_sleep

Synopsis

```
int RT_sleep(double secondes)
```

Include

"rt_interface.h"

Parameters

secondes

delay in seconds

Return value

0 if successful, -1 otherwise

Description

This function suspends the current task for the given time *secondes*. Parameter *secondes* can be less than 1. The effective sleeping time may be different from the given one (see RT-Linux nanosleep()). It uses the nanosleep() RT-Linux function call.

RT_fifo_init

Synopsis

```
int RT_fifo_init(int nb_rtf, int dc_rtf)
```

Include

"rt_interface.h"

Parameters

nb_rtf

number of fifos to create

dc_rtf

first fifo device number to be used

Return value

0 if successful, -1 otherwise

Description

This function creates a fixed number of RT fifos that are used by the Communication channel objects. Communication channels objects are FIFO-style Inter-Task Communication (ITC) mechanisms with blocking read/write features.

The number of created fifos is nb_rtf; it is also the maximum number of communication channels that the application can use. The created fifos are numbered from 0 to nb_rtf-1; this number is different from the fifo device number (eg: the # in /dev/rtf#). Parameter dc_rtf specifies the fifo device number that has to be used for channel number 0, other channels use increasing fifo device numbers (eg: if dc_rtf equals10, channel number 0 uses /dev/rtf10, channel number 1 uses /dev/rtf11, ...). This function has to be called within the init_module() function of the application. This function only exists in combination with RT kernel (such as RT-Linux) that does not allow fifo creation within a RT context. We all expect that this architectural constraint will disappear in future RT-Linux versions.

See also

RT_fifo_open_fd, RT_fifo_open

RT_fifo_uninit

Synopsis

```
int RT_fifo_uninit(void)
```

Include

"rt_interface.h"

Return value

0 if successful, -1 otherwise

Description

This function destroys previously created fifos and has to be called within the cleanup_module of the application.

See also

RT_fifo_init

RT_fifo_open

Synopsis

```
TrCanal RT_fifo_open(int num, in rw)
```

Include

"rt_interface.h"

Parameters

num

Communication channel number

Return value

ID of the newly created channel or NULL if an error occurred.

Description

This function opens a Communication channel based on a previously created RT-fifo (RT_fifo_init() function call)

Communication channels are useful objects providing ITC mechanisms of type FIFO with blocking read and write features.

Parameter num specifies the channel number to be used for the newly created channel object. This number should be less than the maximum number of channels available, as specified at initialisation (see RT_fifo_init()).

See also

RT_fifo_read, RT_fifo_write, RT_fifo_init

RT_fifo_open_fd

Synopsis

```
TrCanal RT_fifo_open_fd(int fd)
```

Include

"rt_interface.h"

Parameters

fd

fifo device number or file descriptor (e.g.: the # in /dev/rtf#)

Return value

ID of the newly created channel or NULL if an error occurred.

Description

This function opens a Communication channel using the pseudo file descriptor fd of an existing system character device. This pseudo file descriptor is the fifo device number (e.g.: the # in /dev/rtf#). The fifo should have been created within the init_module() of the application.

Communication channels are useful objects providing ITC mechanisms of type FIFO with blocking read and write methods.

See also

RT_fifo_read, RT_fifo_write, RT_fifo_close

RT_fifo_fd

Synopsis

```
int RT_fifo_fd(TrCanal c)
```

Include

"rt_interface.h"

Parameters

c

Communication channel

Return value

This function returns the pseudo file descriptor of the character device associated to a Communication channel object. This is the fifo device number of the RT-fifo associated to the channel.

See also

RT_fifo_open_fd, RT_fifo_read, RT_fifo_write, RT_fifo_close

RT_fifo_read

Synopsis

```
int RT_fifo_read(TrCanal c, void *buf, size_t longueur)
```

Include

"rt_interface.h"

Parameters

c

communication channel

buf

buffer where data should be copied to

longueur

number of bytes to be read

Return value

longueur or -1 if an error occurred

Description

This function performs a blocking read on the specified communication channel. RT_fifo_read() does not return until the specified data size has been read. The calling task is suspended during function blocking.

RT_fifo_read uses a fifo-handler and a synchronization semaphore in order to make the function blocking: function sem_wait() the semaphore while fifo handler sem_post() it as soon as a data is available in the fifo.

See also

RT_fifo_open_fd, RT_fifo_write

RT_fifo_write

Synopsis

```
int RT_fifo_write(TrCanal c, const void *buf, size_t longueur)
```

Include

"rt_interface.h"

Parameters

c

communication channel

buf

buffer to copy data from

longueur

number of bytes to write

Return value

longueur or -1 if an error occurred

Description

This function writes data to the specified communication channel; the function does not return until the full data size has been written to the fifo (eg: exactly as RT-Linux does). Although, there is no way to know when the task/process that is reading the fifo effectively read sent data.

See also

RT_fifo_open_fd, RT_fifo_read

RT_fifo_close

Synopsis

```
void RT_fifo_close(TrCanal c )
```

Include

"rt_interface.h"

Parameters

c

communication channel

Description

This function close communications on specified channel and frees allocated resources.

This function should never be used on pre-allocated communication channels such as the one provided by RT_fifo_open().

See also

RT_fifo_open_fd, RT_fifo_read, RT_fifo_write

2.4 Implementation issues

None

2.5 Tests and validation

Unit test application provided with component and validation through Robotic application.

2.5.1 Validation criteria

Unit test application success and Robotic application behavior as expected.

2.5.2 Tests

See unit test application in the component folder.

2.5.3 Results and comments

Component successfully tested.

2.6 Examples

See unit test application in the component folder.

2.7 Installation instructions

Do not need any installation.

Chapter 3. UNIX compatibility headers set

3.1 Summary

Name:

UNIX compatibility header set

Description:

Header file set for UNIX-compliant application portability improvement.

Author:

F. Russotto

Reviewer:

J. Brisset

Layer:

Application level

Version:

0.1

Status:

Stable

Dependencies:

rtl, rtl_sched, ftl_fifo, rtl_malloc, onetd.

Release date:

MS 5

3.2 Description

The C header files in this set define useful constants, macros, types, etc… with partial compliance with the Standard UNIX Specification (SUS) API and header file tree. Including these files in the applications improve portability from a standard UNIX Operating System to a Linux Real-Time kernel such as RTLinux/OCERA, by providing a UNIX-like API on top of system/kernel services. One should take care that definitions in these files generally do not provide the standard UNIX services that may be expected from such API, but rather some system/kernel limited ones, which should though satisfy basic requirements like printing strings to the screen, allocating memory, or using IP sockets. Therefore, the provided services may lead to slightly or even very different behaviors regarding the standard UNIX ones. Using such services requires developers to know their limitations.

3.3 File List

Here is a list of all documented files with brief descriptions:

assert.h (Basic diagnostic utilities with an ISO C99 7.2 interface)
errno.h (Basic Errno definitions with an ISO C99 7.5 interface)
netdb.h (Basic definitions for network database operations (POSIX interface))
stdio.h (Basic input/output services with an ISO C99 7.19 interface)
stdlib.h (Basic general utilities with an ISO C99 7.20 interface)
arpa/inet.h (Basic definitins for internet operations (POSIX interface))
netinet/in.h (Basic Internet Protocol family definitions with POSIX interface)
rpc/rpc.h (Basic Sun Microsystems RPC services)
rpc/types.h
rpc/xdr.h
sys/select.h (Definition of the select() POSIX function)
sys/socket.h (Basic Internet Protocol family socket services with POSIX interface)

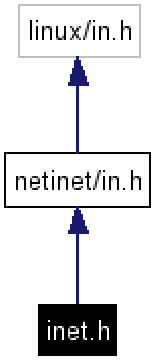
---

arpa/inet.h

Basic definitions for internet operations (POSIX interface).

```
#include <netinet/in.h>
```

Include dependency graph for inet.h:



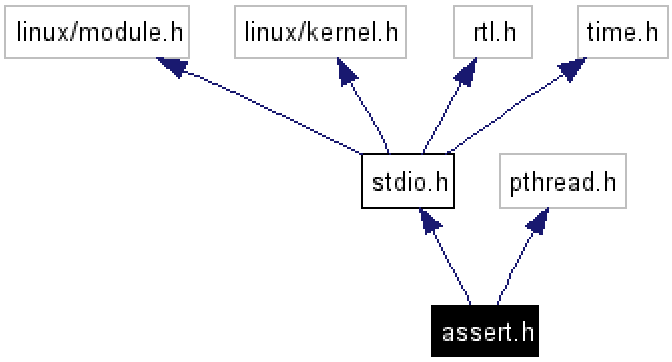
---

assert.h

Basic diagnostic utilities with an ISO C99 7.2 interface.

```
#include <stdio.h>
#include <pthread.h>
```

Include dependency graph for assert.h:



Defines

#define **__ASSERT_FUNCTION** ((__const char *) 0)
#define **__ASSERT_PROCESS** ((unsigned int)pthread_self())
#define **assert**(cond)

Define Documentation

#define assert(cond)

```
Value:if (!(cond)) { \
                fprintf(stderr, "0x%x: %s:%d: %s: Assertion failed. Aborting rt-thread\n",
\
                    __ASSERT_PROCESS, \
                    __FILE__, __LINE__, \
                    __ASSERT_FUNCTION); \
                pthread_exit(NULL); }
```

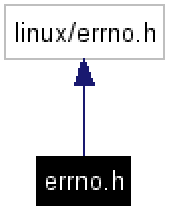
---

errno.h

Basic Errno definitions with an ISO C99 7.5 interface.

```
#include <linux/errno.h>
```

Include dependency graph for errno.h:



Defines

#define **ENOTSUP** EOPNOTSUPP
#define **__set_errno**(x) do { errno = (x); } while (0)
#define **__reterror**(x, y) do { errno = (x); return y; } while (0)

Variables

int **errno**

---

netdb.h

Basic definitions for network database operations (POSIX interface).

Data Structures

struct hostent

Description of data base entry for a single host.

Defines

#define h_addr h_addr_list[0]

Define Documentation

#define h_addr h_addr_list[0]

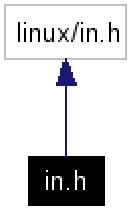
Address, for backward compatibility.

Netinet/in.h

Basic Internet Protocol family definitions with POSIX interface.

```
#include <linux/in.h>
```

Include dependency graph for in.h:



Typedefs

typedef uint32_t **in_addr_t**

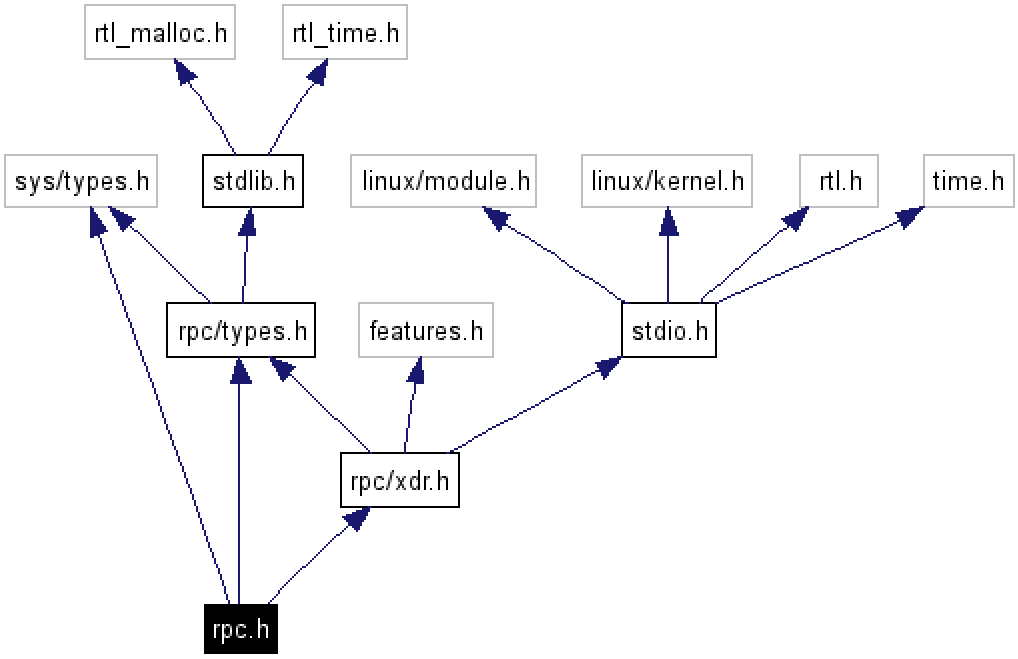
---

rpc/rpc.h

Basic Sun Microsystems RPC services.

```
#include <sys/types.h>
#include <rpc/types.h>
#include <rpc/xdr.h>
```

Include dependency graph for rpc.h:



Typedefs

typedef long long int **quad_t**

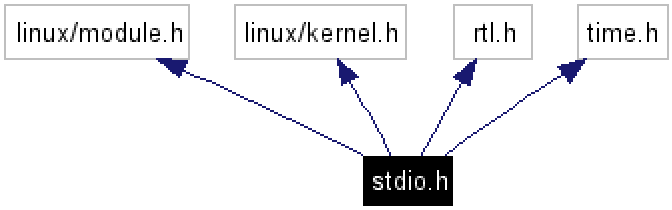
---

stdio.h

Basic input/output services with an ISO C99 7.19 interface.

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <rtl.h>
#include <time.h>
```

Include dependency graph for stdio.h:



Defines

#define ***osprintf*** *rtl*printf
#define **stdin** ((FILE*)0x0)
#define **stdout** ((FILE*)0x1)
#define **stderr** ((FILE*)0x2)
#define **EOF** (-1)
#define **printf** *os*printf
#define **puts**(s) printf("%s\n", s)
#define **fopen**(fn, m) ((FILE*)fn)
#define **fclose**(fd) ((void)0)
#define **fprintf**(fd, fmt, args…) printf(fmt, ##args)
#define **perror**(s) fprintf(stderr, "error %d: %s\n", errno, s)
#define **fputs**(str, fd) fprintf(fd, "%s", str)
#define **__fncall** ((__const char *) 0)
#define **fscanf**(stream, fmt, args…) (EOF+0*printf("WARNING: %s:%d: %s: fscanf(\"0x%x\"): service not supported\n", __FILE__, __LINE__, __fncall, (int)(stream)))
#define **fgets**(string, size, stream) (NULL+0*printf("WARNING: %s:%d: %s: fgets(\"0x%x\"): service not supported\n", __FILE__, __LINE__, __fncall, (int)(stream)))
#define **getc**(stream) (EOF+0*printf("WARNING: %s:%d: %s: getc(\"0x%x\"): service not supported\n", __FILE__, __LINE__, __fncall, (int)(stream)))
#define **dprintf**(fmt, args...) printf(fmt, ##args)

Typedefs

typedef void **FILE**

Variables

int **errno**

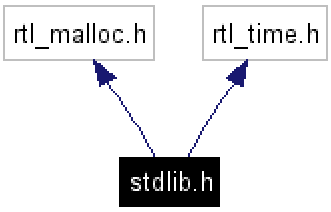
---

stdlib.h

Basic general utilities with an ISO C99 7.20 interface.

```
#include <rtl_malloc.h>
#include <rtl_time.h>
```

Include dependency graph for stdlib.h:



Defines

#define **srand**(n) ((void)0)
#define **RAND_MAX** (0x7FFFFFFF)
#define **rand**() ((int)(gethrtime()&(hrtime_t)RAND_MAX))
#define **exit**(n) { rtl_printf("0x%p: exiting with code: %d\n", pthread_self(), n); pthread_exit((void*)(n)); }
#define ***osprintf*** *rtl*printf
#define **__fncall** ((__const char *) 0)
#define **getenv**(name) (NULL+0*_os_printf("WARNING: %s:%d: %s: getenv(\"%s\"): service not supported\n", __FILE__, __LINE__, __fncall, name))

#define **system**(cmd) (-1+0*_os_printf("WARNING: %s:%d: %s: system(\"%s\"): service not supported\n", __FILE__, __LINE__, __fncall, cmd))

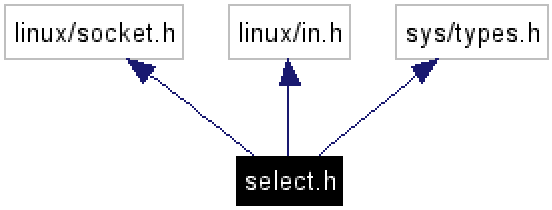
---

sys/select.h

Definition of the select() POSIX function.

```
#include <linux/socket.h>
#include <linux/in.h>
#include <sys/types.h>
```

Include dependency graph for select.h:



Defines

#define **__fncall** ((__const char *) 0)
#define **select**(n, rfds, wfds, efds, to) (-1+0*printf("WARNING: %s:%d: %s: select(): function call not supported\n", __FILE__, __LINE__, __fncall)+0*usleep((to)->tv_sec*1000000+(to)->tv_usec))

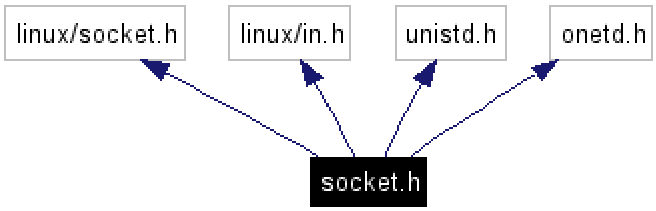
---

sys/socket.h

Basic Internet Protocol family socket services with POSIX interface.

```
#include <linux/socket.h>
#include <linux/in.h>
#include <unistd.h>
#include <onetd.h>
```

Include dependency graph for socket.h:



Defines

```
#define close_socket  ocn_close
#define socket  ocn_socket
#define bind  ocn_bind
#define recvfrom(s, b, l, f, ad, al)  ocn_recvfrom(s, b, l, f, ad, *(al))
#define sendto  ocn_sendto
```

3.4 Implementation issues

None

3.5 Tests and validation

Robotic application validation.

3.5.1 Validation criteria

Robotic application behaves as expected.

3.5.2 Tests

Robotic application execution.

3.5.3 Results and comments

Robotic application works as expected.

3.6 Examples

None

3.7 Installation instructions

Add the headers set path to the include directories in the compiler command-line options (e.g.: $(CC) … -I…/include …).

Chapter 4. hapticctrl component

4.1 Summary

Name:

hapticctrl

Description:

Controller application kernel-module top-level component.

Author:

F. Russotto

Reviewer:

J. Brisset

Layer:

Application level

Version:

2.2.24

Status:

Stable

Dependencies:

rtl, rtl_sched, ftl_fifo, rtl_malloc, rt_interface, horloge, other generic RT components, all generic robotics components, all generic haptics components, protocol.

Release date:

MS 5

4.2 Description

This component is the top-level component of the OCERA Robotic application kernel-module. It includes all application-specific definitions and functions, including:

- platform-specific definitions and functions (e.g.: init_module, cleanup_module, …)
- the main application initialisation function (hapticctrl)
- The Virtuose robotics arm simulator

4.3 API / compatibility

`init_module`

Application kernel module initialization function

`cleanup_module`

Application kernel module cleanup function

hapticctrl
Entry point of the application initialization task (INIT)

_servo
Entry point of the main servo-control task (SERVO)

---

init_module

Synopsis

```
int init_module(void)
```

Return value

Always 0

Description

This function is the application kernel module initialization function. The only task carried out by the init_module() function is the following:

- Create the application initialisation thread (INIT). All further application initialisations are carried out by the INIT thread.

Most application initializations are made in the hapticctrl() function (which is the entry point of the INIT thread). The INIT task will start all other needed tasks of the application.

See also

hapticctrl, cleanup_module

---

cleanup_module

Synopsis

```
void cleanup_module(void)
```

Description

This function is the application kernel module cleanup function. The cleanup_module() function terminates all running tasks and exits the kernel.

See also

init_module, hapticctrl

---

hapticctrl

Synopsis

```
int hapticctrl(char * robot)
```

Return value

0 if initialization is successful, -1 otherwise

Parameters

robot

Robot name (in human readable format)

Description

This function is the entry point of the application initialisation task (INIT). Hapticctrl() function (mainly) performs the following steps:

- initialise haptics & robotics components
- initialise protocol component (this creates the PROTO_IN task)
- create the main and the degraded servo-control tasks using OCERA fault-tolerance component API (SERVO, SERVO_DGD).

See also

_servo

_servo

Synopsis

```
void _servo(void)
```

Description

This function is the entry point of the main servo-control task (SERVO). _servo() performs the following jobs, in a periodic manner (period = 1 ms):

- sample Virtuose robot-arm position and speed from the input boards (joint space)
- compute robot-arm position and speed in Cartesian space
- compute robot-arm / virtual-object coupling torque in Cartesian space using virtual object position and speed received from client application (received by PROTO_IN task and stored in a shared buffer protected by a MUTEX semaphore)
- compute requested coupling torque in joint space (using the robot Jacobian matrix)
- compute robot-arm servo-control commands from requested coupling torque
- send computed commands to the robot –arm output boards
- send back coupling torque to the client application (using ONetD socket)

See also

hapticctrl

4.4 Implementation issues

Not applicable

4.5 Tests and validation

Robotic application validation.

4.5.1 Validation criteria

Robotic application behaves as expected.

4.5.2 Tests

Robotic application execution.

4.5.3 Results and comments

Robotic application works as expected.

4.6 Examples

None

4.7 Installation instructions

Not applicable

Chapter 5. Conclusion and future works

The OCERA Robotic application version 2 has been strongly consolidated, compared to version 1. The application core modules have been totally ported once again to RTLinux/OCERA, improving architecture and performance. OCERA fault tolerance has been integrated to the application, and the communication is now based on a kernel implementation of a socket API (OCERA ONetD component). As a result, application v2 is now much more stable and reliable, and it runs with very good performance on hardware.

Several tests of the application have been made using RTLinux/OCERA[4]. All tests showed that the application performed at the same level of performance as when using VxWorks real-time kernel. Only the networking component showed some limitations when hardly solicited (see below).

Working on top of Linux sockets layer and network drivers, the OCERA networking component (ONetD) provide a simple, flexible and efficient solution for IP networking communication. When involved in a hard real-time context, however, ONetD showed limited performance due to the use of the Linux services. To bring performance to perfection, a pure kernel based networking component would be appreciated in some future developments based on OCERA.

OCERA Fault-Tolerance components provide an innovative and efficient solution for application faults management. Although lacking memory fault management (which would require memory fault handling from low-level kernel layers), these components offer smart features to allow application fallback in case of unexpected crash or deadlock situations.

To be added to the drawbacks table, the RTLinux/OCERA framework is lacking efficient debugging tools. Despite very efficient trace tools such as OCERA POSIX Trace, debugging huge applications may be really painful when kernel crash occur. This drawback could be circumvented by providing (for instance) either a RTLinux/OCERA API working on top of user-space Linux (to be used during tests of application with degraded RT performances) or, (much better) efficient memory fault and exception handling in order to avoid kernel crash.

To conclude on the work done in WorkPackage 9, RTLinux/OCERA real-time kernel and OCERA components provided a robust, almost complete and high performance system layer, making OCERA package a serious open-source alternative to commercial products such as VxWorks. HAPTION, which distributes the Virtuose products, expressed a strong interest towards WP9 results and intends to distribute a Virtuose product based on the OCERA package in the near future.

---

[4] RTLinux 3.2-pre1 OCERA patched kernel

Chapter 6. Table of acronyms

ACPI	Advanced Configuration and Power Interface
API	Application Programming Interface
APIC	Advanced Programmable Interrupt Controller
APM	Advanced Power Management
CRC	Circular Redundancy Checksum
D	Dimensions
DMA	Direct Memory Access
DoF	Degree(s) of Freedom
FIFO	First In First Out
GUI	Graphical User Interface
IP	Internet Protocol
ITC	Inter-Tasks Communication
MUTEX	MUTual-Exclusion semaphore
OCERA	Open Components for Embedded Real-time Applications
PCI	Peripheral Component Interconnect
POSIX	Portable Operating System Interface
RT	Real-Time
SUS	Standard Unix Specification
UDP	User Datagram Protocol
XDR	eXternal Data Representation (Sun microsystems)