WP9 – Validation on platform

Deliverable D9rb.3_rep – Robotic Application Development Report

WP9 – Validation on platform : Deliverable D9rb.3_rep – Robotic Application Development Report

by F. RUSSOTTO (CEA)

Published: november 2003

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	1/10/2003	First release

Chapter 1. Introduction

1.1 Summary

This document is the development report of the Robotic application V1. Robotic application is intended to demonstrate efficiency of Linux, RT-Linux and the system components developed within 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 consists in a porting of an existing industrial robotic application which is to big to be fully detailed here; 2) application has to be considered as a validation tool within the OCERA project, describing the whole application is then off topic. That way, only the general application functioning and application components that interface with Linux, RT-Linux and OCERA system components will be described in details in this document.

1.2 Description

The Robotic application that has been developed is a Servo-control application of a master robot arm called: Virtuose. Virtuose robot arm is intended to be used in an 3D virtual environment consisting of screen displaying a 3D virtual scene made of 3D objects. Virtuose robot arm allows user to manipulate virtual objects of the 3D scene with 6 DoF force feedback so that user can feel weight, inertia, collisions between objects as if he manipulated real objects.

Figure 1 : Virtuose robot arm in action

For application V1, the developed servo-control application is not connected to a real Virtuose robot arm. A Virtuose robot simulator has been developed instead in order to test correct behavior of the application prior to use it on a real hardware robot.

The robotic application actually consists in two applications, a Controller application and a Client application, running on separate machines connected together by Ethernet.

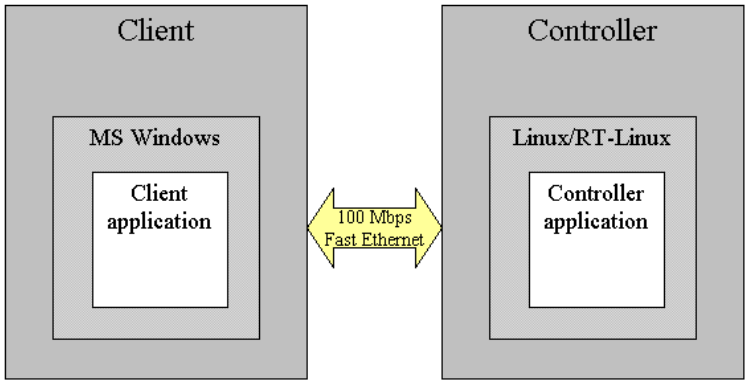

Figure 2: Client - Controller general view

Description of each application follows. OCERA is mainly concerned by the Controller application, running the robotic control (and hard-real time) part of the application.

1.2.1 Client application

The client application is a Win32 executable running on a Windows desktop computer. Using standard Ethernet connection, the client application can communicate with the controller application. Data transmitted are encapsulated into a specific protocol working on top of UDP/IP. Data transmitted is encoded to then decoded from the Sun® XDR® format to avoid data corruption due to communication between different platforms.

The client application includes a simplified Physics/Dynamics engine that is used to simulate the behavior of a virtual scene consisting of two half-rigid spheres; one of these spheres is fixed in the scene while the other one is mobile. A GUI represents the virtual environment and allows user to apply (virtually) forces and torques on the mobile sphere.

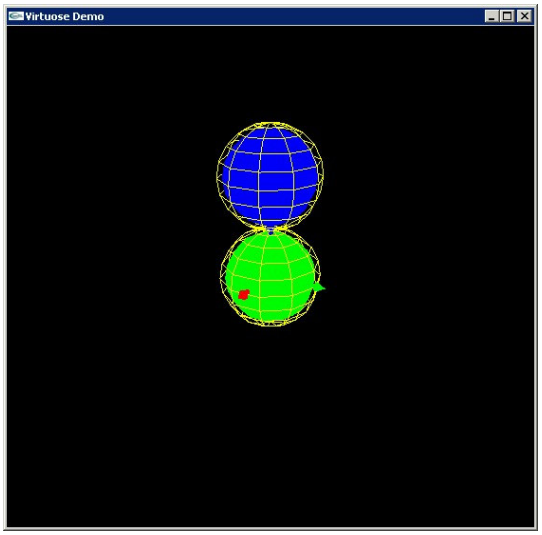

Figure 3: Client application GUI

When the Client and the Controller applications are running and connected together through Ethernet, the mobile sphere and the (simulated) Virtuose robot arm handle are bi-laterally coupled, so that:

- any force (virtually) applied on the mobile sphere (from the Client) involves movement of both the mobile sphere and the (simulated) Virtuose robot arm,
- any force (virtually) applied on the (simulated) Virtuose robot arm handle (from the Controller) involves movement of both the mobile sphere and the (simulated) Virtuose robot arm.

1.2.2 Controller application

The Controller application consists in three executives: one Linux kernel module and two user-space processes. The kernel module runs the robotic control (and hard RT) part of the application, while the user-space (soft RT) processes are respectively the Client / Controller communication interface (Ethernet/UDP/IP) and the Simulated Virtuose robot user interface allowing user to act on simulated Virtuose robot.

1.2.2.1 Robotic control (kernel module – hard RT)

Robotic control kernel module application includes several components divided into the three main following categories. Most of these components will not be described in details in this document because they are robotics or haptics[1] related and do not have any interface with the OS (which is the center of interest in the OCERA project); they are briefly presented in this section to provide user some information about how the application works.

- Generic Real-Time components: these components provide an API that encapsulates common features of the RT kernel. They can be used within the context of any real-time application in order to enhance the application portability towards several RT kernel or OS. Most of these components strongly interface with RT-Linux and OCERA components and the main one will be fully described in the following sections.
- Generic robotics components: theses components provide common robotics functions that can be used in the context of any robotics application (including haptics). These components use the generic RT components but do not have any other interface with the OS and so will not be detailed.
- Haptics components: these components provide haptic specific features to the application. Except the “protocole” and the “simul” components which interfaces with RT-Linux, none of these components will be described in this document.

[1] Haptics is related to tactile robotics

1.2.2.1.1 Generic Real-Time components

The following figure shows the dependency tree of the Generic RT components. Components represented in yellow are native RT-Linux components, pink ones are OCERA components and white ones are application components.

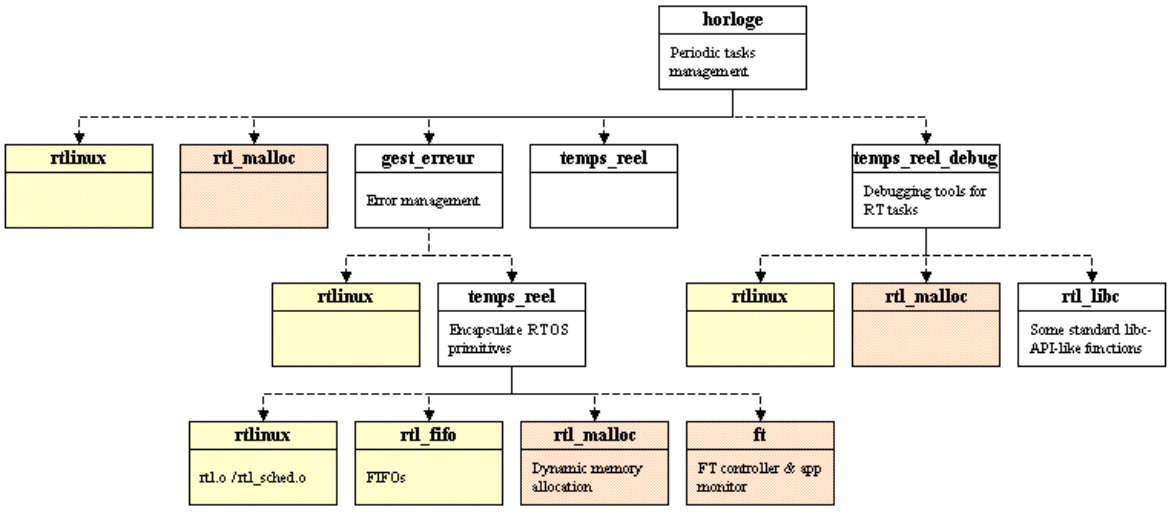

Figure 4: Generic Real-Time components dependency tree

Components brief description follows:

- **temps_reel** [2] : this component encapsulates mostly used RT kernel primitives such as: task management and ITC; it is fully detailed in Chapter 2.
- **horloge** [3] : this component provide a way to manage a set of periodic tasks synchronized on a same common clock; component is fully detailed in Chapter 3.
- gest_erreur [4] : this component provides common features for application error management; component has still no interface with RT-Linux or other system component (but will have in future versions), therefore it is not described in the following sections.
- rtl_libc : this component is intended to provide a little libc-like API for screen I/O or file access within RT tasks; this component is not implemented yet but will be in future versions, therefore it is not described in the following sections.
- temps_reel_debug : this component is intended to provide an interface to a RT trace toolkit such as POSIX trace; this component is not implemented yet but will be in future versions, therefore it is not described in the following sections.

---

[2] temps-reel stands for : real-time

[3] horloge stands for : clock

[4] gestion d'erreur stands for : error management

1.2.2.1.2 Robotics, haptics and application-specific components

The following figure shows the simplified dependency tree of the overall robotic control kernel-module (hapticctrl). Components represented in yellow are native RT-Linux components, pink ones are OCERA components and white ones are application components.

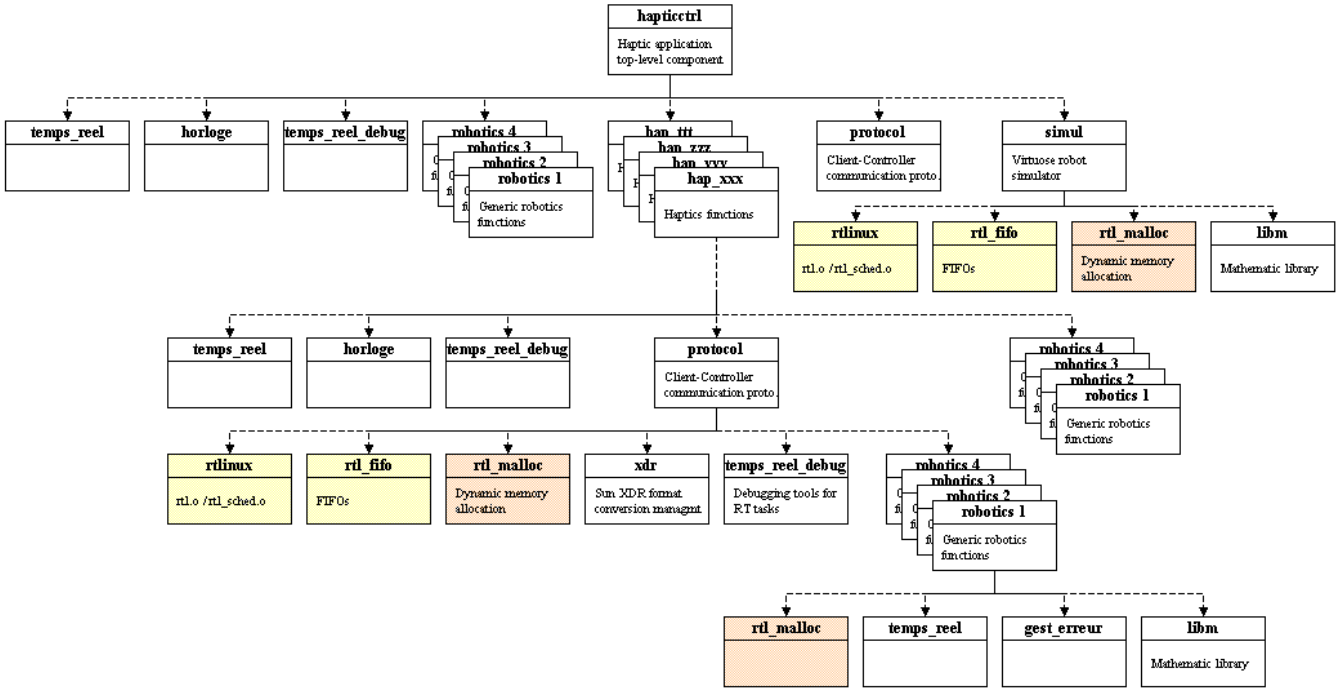

Figure 5: Robotic control kernel-module dependency tree

Components brief description follows:

- 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 provide a simple communication protocol API to be used by the kernel-module to communicate with the Client application; component is described in Chapter 4.
- **hapticctrl** : this component is the kernel-module top-level component; it includes all application specific definitions; component is described in Chapter 5.
- **simul** : this component is the Virtuose robot arm simulator; component has been merged (as a sub-component) with the hapticctrl component and is described in the same section as hapticctrl component (Chapter 5.).

1.2.2.2 User-space interfaces

1.2.2.2.1 Communication protocol interface (proto_interface)

This component is a Linux user-space executable connected to the kernel-module protocol component through RT-fifos (see §1.2.2.2.1 and Chapter 4.). It provides to Controller application a standard Linux socket interface to communicate with the Client application using UDP/IP. This component is described in Chapter 6.

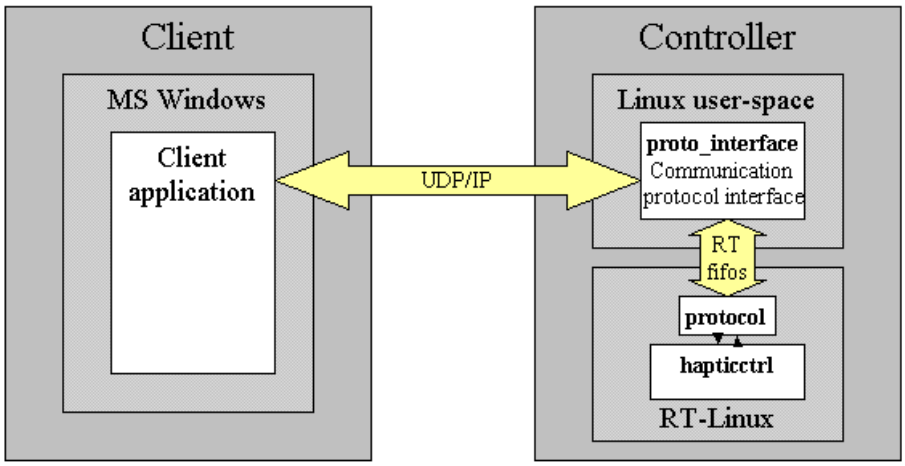

Figure 6 : proto_interface component principle of functioning

1.2.2.2.2 Virtuose robot arm simulator user interface

This component is a Linux user-space executable connected to the kernel-module hapticctrl/virtsim component (see §1.2.2.1.2 and Chapter 5.). It provides a user interface to interact with Virtuose robot arm simulator component (hapticctrl/virtsim). This component is described in Chapter 7.

vitsim component includes an implementation of a GUI based on QT and Glut allowing a pretty nice visual monitoring of the simulated Virtuose robot arm. Unfortunately, due to some Linux/RT-Linux problems, the GUI implementation could not be successfully connected to the application kernel-module; we hope being able to do so as soon as related Linux/RT-Linux problems are solved.

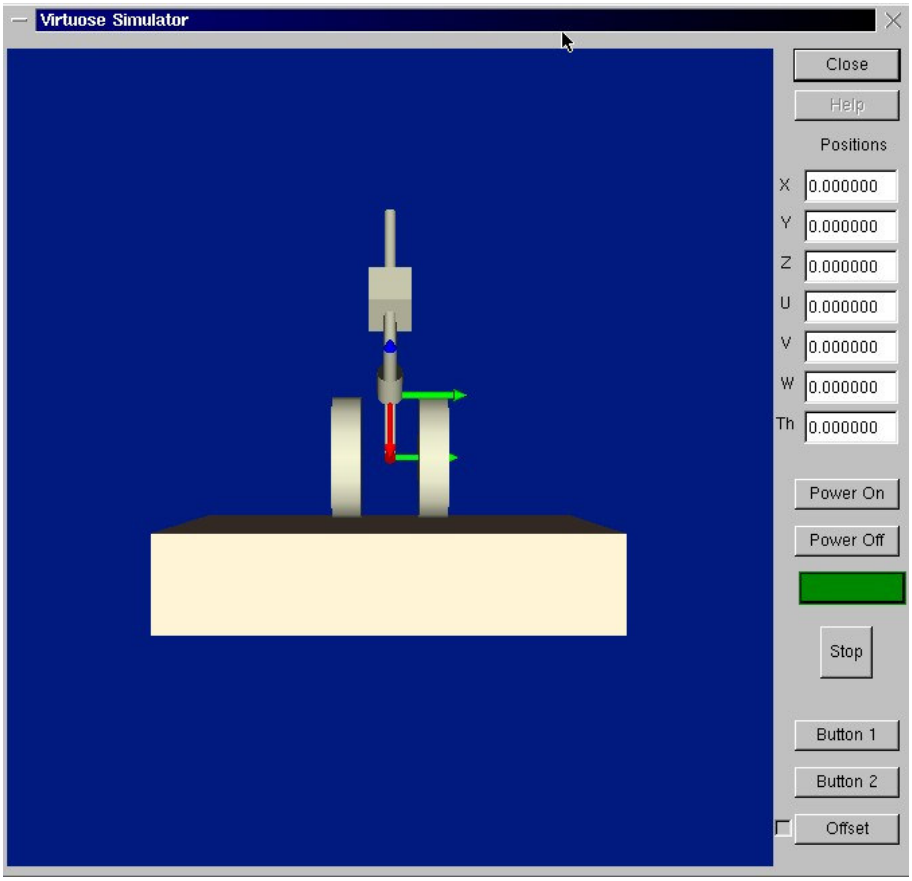

Figure 7 : Virtuose robot arm QT/Glut user interface

Chapter 2. temps_reel component

2.1 Summary

Name:

temps_reel (stands for: real-time)

Description:

Encapsulates common RT kernel primitives for application portability enhancement

Author:

F. Russotto (russotto@cea.fr)

Reviewer:

Layer:

Application level

Version:

0.1 beta

Status:

Design

Dependencies:

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

Release date:

MS 4

2.2 Description

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

2.3 API / compatibility

`TR_creer_semaphore_synchro`
Create a synchronization semaphore

`TR_creer_semaphore_mutex`
Create a MUTEX semaphore

TR_detruire_semaphore
: Destroy a synchronization or MUTEX semaphore

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

TR_attendre_semaphore
: Lock / wait a semaphore (blocking)

TR_poster_semaphore
: Unlock / post a semaphore

TR_creer_tache
: Create a Real-Time task / thread

TR_detruire_tache
: Destroy (cancel) a RT task

TR_delai
: Suspend the current task for a given delay

TR_creer_watch_dog
: watchdog (timer) creation

TR_lancer_watch_dog
: Arm a watchdog (timer)

TR_desarmer_watch_dog
: De-arm a watchdog (timer)

TR_detruire_watch_dog
: Delete a watchdog (timer)

TR_initialiser_es
: Initialize communication channels

TR_terminer_es
: Uninitialize communication channels

TR_creer_canal_rtfifo
: Create a communication channel from a channel number

TR_creer_canal
: Create a communication channel from a file descriptor

TR_descripteur_canal
: Return file descriptor associated to a communication channel

TR_lire_canal
: Read data from a communication channel (blocking)

TR_ecrire_canal
: Write data to a communication channel

TR_fermer_canal
: Close communications on a channel

TR_creer_semaphore_synchro

Synopsis

```
TrSem TR_creer_semaphore_synchro(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

TR_creer_semaphore_mutex, TR_detruire_semaphore, TR_attendre_semaphore, TR_poster_semaphore

---

TR_creer_semaphore_mutex

Synopsis

```
TrSem TR_creer_semaphore_mutex(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

TR_creer_semaphore_synchro, TR_detruire_semaphore, TR_attendre_semaphore, TR_poster_semaphore

---

TR_detruire_semaphore

Synopsis

```
int TR_detruire_semaphore(TrSem sem)
```

Include

"tr.h"

Parameters

sem

semaphore to be deleted

Return value

0 if succesfull, -1 otherwise.

Description

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

See also

TR_creer_semaphore_synchro, TR_creer_semaphore_mutex

TR_tester_semaphore

Synopsis

```
int TR_tester_semaphore(TrSem 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 in use by another task (locked or waited) then the function returns -1 immediately without blocking; if sem is not in use by another task, then function returns 0 and sem becomes locked/waited.

See also

TR_creer_semaphore_synchro, TR_creer_semaphore_mutex

TR_attendre_semaphore

Synopsis

```
int TR_attendre_semaphore(TrSem 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. This uses either the sem_wait() or the pthread_mutex_lock() functions depending on the sempahore type. If sem is in use by another task (locked or waited) then function blocks undefinitely waiting for semaphore unlocking/posting (calling task is suspended); as soon as sem is unlocked/posted (or if sem was not in use), then TR_attendre_semaphore() returns 0, sem becomes locked/waited and the calling task is woked up if possible. TR_attendre_semaphore() can be used either to grant access to a shared ressource or to suspend current task for syncrhonization by another task.

See also

TR_creer_semaphore_synchro, TR_creer_semaphore_mutex

TR_poster_semaphore

Synopsis

```
int TR_poster_semaphore(TrSem 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 ressource.

See also

TR_creer_semaphore_synchro, TR_creer_semaphore_mutex

TR_creer_tache

Synopsis

```
TrTache  TR_creer_tache(const  char  *nom,  unsigned  int  priorite,  size_t
                        taillePile, (void *)(*ptEntree)(void *), void *arg)
```

Include

"tr.h"

Parameters

nom

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

priorite

priority of the task (0 is the highest)

taillePile

task stack size

ptEntree

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 priorite should be given according to the following standard: 0 is the highest priority and sched_get_priority_max(SCHED_FIFO) is the lowest one.

TR_creer_tache() performs the following steps:

- initialize pthread attributes using parameters ptEntree and arg
- set pthread scheduling attributes using priority = sched_get_priority_max(SCHED_FIFO) - priorite
- 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 taillePile
- set pthread detachstate using PTHREAD_CREATE_DETACHED
- create the RT task using standard pthread_create()

See also

TR_redemarrer_tache, TR_detruire_tache

TR_detruire_tache

Synopsis

```
int TR_detruire_tache(TrTache tache)
```

Include

"tr.h"

Parameters

tache

task to be deleted (canceled)

Return value

0 if successful, -1 otherwise

Description

This function destroys (cancels) the specified Real-Time task and frees allocated memory ressource. This uses standard pthread_cancel() and free() provided by the rtl_malloc OCERA component.

See also

TR_creer_tache, TR_redemarrer_tache

TR_delai

Synopsis

```
int TR_delai(double secondes)
```

Include

"temps_reel.h"

Parameters

secondes

suspension time 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()). This uses nanosleep() RT-Linux function call.

TR_creer_watch_dog

Synopsis

```
TrWd TR_creer_watch_dog(void)
```

Include

"temps_reel.h"

Return value

ID of the newly created watchdog (timer) created

Description

This function (not implemented yet) will create a watchdog (timer)

See also

TR_lancer_watch_dog, TR_desarmer_watch_dog, TR_detruire_watch_dog

---

TR_lancer_watch_dog

Synopsis

```
int TR_lancer_watch_dog(TrWd wd, double secondes, void *(*fnFinDelai)(void
*), void *arg)
```

Include

"temps_reel.h"

Parameters

wd

watchdog (timer) ID

seconde

watchdog (timer) fire delay (in seconds)

fnFinDelai

pointer to a call-back handler

arg

arguments to be passed to the call-back handler

Return value

0 if successful, -1 otherwise

Description

This function (not implemented yet) will arm the given watchdog (timer). Parameter fnFinDelai specifies a pointer to a call-back function (handler) to be fired (called) at watchdog (timer) expiration.

See also

TR_creer_watch_dog, TR_desarmer_watch_dog, TR_detruire_watch_dog

---

TR_desarmer_watch_dog

Synopsis

```
int TR_desarmer_watch_dog(TrWd wd)
```

Include

"temps_reel.h"

Parameters

wd

watchdog ID

Return value

0 if successful, -1 otherwise

Description

This function (not implemented yet) will de-arm a previously armed watchdog (timer). De-arming a watchdog (timer) prevents the call-back function to be fired (called).

See also

TR_lancer_watch_dog, TR_detruire_watch_dog

TR_detruire_watch_dog

Synopsis

```
int TR_detruire_watch_dog(TrWd wd)
```

Include

"temps_reel.h"

Parameters

wd

watchdog (timer) ID

Return value

0 if successful, -1 otherwise

Description

This function destroys a previously created watchdog (timer).

See also

TR_creer_watch_dog, TR_lancer_watch_dog, TR_desarmer_watch_dog, TR_detruire_watch_dog

---

TR_initialiser_es

Synopsis

```
int TR_initialiser_es(int nb_rtf, int dc_rtf)
```

Include

"temps_reel.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=10, 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 for RT kernel (such as RT-Linux) that do not allow fifo creation within a RT context; we all expect that this architectural constraint will disappear in future RT-Linux versions.

See also

TR_creer_canal, TR_creer_canal_rtfifo

TR_terminer_es

Synopsis

```
int TR_terminer_es(void)
```

Include

"temps_reel.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

TR_initialiser_es

TR_creer_canal_rtfifo

Synopsis

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

Include

"temps_reel.h"

Parameters

num

Communication channel number

Return value

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

Description

This function creates a Communication channel based on a previously created RT-fifo (TR_initialiser_es() 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 initialization (see TR_initialiser_es()).

See also

TR_lire_canal, TR_ecrire_canal, TR_initialiser_es

TR_creer_canal

Synopsis

```
TrCanal TR_creer_canal(int fd)
```

Include

"temps_reel.h"

Parameters

fd

fifo device number or file descriptor (eg: the # in /dev/rtf#)

Return value

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

Description

This function creates a Communication channel using the file descriptor fd of an existing system character device; this file descriptor is the fifo device number (eg: 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

TR_lire_canal, TR_ecrire_canal, TR_fermer_canal

TR_descripteur_canal

Synopsis

```
int TR_descripteur_canal(TrCanal c)
```

Include

"temps_reel.h"

Parameters

c

Communication channel

Return value

This function returns the 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

TR_creer_canal, TR_lire_canal, TR_ecrire_canal, TR_fermer_canal

---

TR_lire_canal

Synopsis

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

Include

"temps_reel.h"

Parameters

c

communication channel

buf

buffer to copy data 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. TR_lire_canal() does not return until the specified data size has been read; the calling task is suspended during function blocking.

TR_lire_canal 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

TR_creer_canal, TR_ecrire_canal

TR_ecrire_canal

Synopsis

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

Include

"temps_reel.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

TR_creer_canal, TR_lire_canal

TR_fermer_canal

Synopsis

```
void TR_fermer_canal(TrCanal c )
```

Include

"temps_reel.h"

Parameters

c

communication channel

Description

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

This function should never be used on prealocated communication channels such as the one provided by TR_creer_canal_rtfifo().

See also

TR_creer_canal, TR_lire_canal, TR_ecrire_canal

2.4 Implementation issues

Not applicable

2.5 Tests and validation

Unit test application provided with component and Robotic application validation.

2.5.1 Validation criteria

Unit test application success and Robotic application behavior as expected.

2.5.2 Tests

See unit test application.

2.5.3 Results and comments

Component successfully tested.

2.6 Examples

None

2.7 Installation instructions

Not applicable

Chapter 3. horloge component

3.1 Summary

Name:

horloge (stands for: clock)

Description:

Synchronous periodic tasks management

Author:

F. Russotto (russotto@cea.fr)

Reviewer:

Layer:

Application level

Version:

0.1 beta

Status:

Design

Dependencies:

rtl, rtl_sched, ftl_fifo, rtl_malloc, temps_reel, gest_erreur, temps_reel_debug

Release date:

MS 4

3.2 Description

This component allows to create periodic tasks synchronized on a common clock. The component initialization function (HORLOGE_initialiser()) creates 2 empty lists of functions and starts a main periodic task named HORLOGE acting as a scheduler.

Using the HORLOGE_ajouter_fct_periodique() primitive, the user application can create a periodic task and attach it to the horloge component so that it is synchronized on the same clock period. All attached periodic tasks necessarily have a period multiple of the main HORLOGE task period. To avoid unexpected context task switch, periodic tasks are created with an effective priority decreased by the main HORLOGE task priority.

The HORLOGE task also performs deadline miss tests on attached tasks and switches from normal behavior to degraded behavior if needed (this job will be performed by the OCERA Fault-Tolerant controler in future versions); for each periodic task, a normal behavior and a degraded behavior have to be defined using two different functions.

3.3 API / compatibility

HORLOGE_initialiser
: Initialize horloge component

HORLOGE_terminer
: Uninitialize horloge component

HORLOGE_ajouter_fct_periodique
: Attach a periodic task

HORLOGE_activer_fct_periodique
: Activate/deactivate a periodic task

HORLOGE_supprimer_fct_periodique
: Detach a periodic task

HORLOGE_forcer_execution
: Immediate execution of a periodic task

HORLOGE_arreter
: Stop the main HORLOGE clock

HORLOGE_redemarrer
: Restart the main HORLOGE clock

_point_d_entree
: Generic entry point for the normal behavior periodic tasks (private)

_horloge_gerer_depassement
: Generic entry point for the degraded behavior periodic tasks (private)

_traiter_top_horloge
: Entry point of the main HORLOGE task (private)

HORLOGE_initialiser

Synopsis

```
int HORLOGE_initialiser(double periodeBase)
```

Include

"horloge.h"

Parameters

periodeBase

: period of the HORLOGE task (in seconds)

Return value

0 if successful, -1 otherwise

Description

This function creates the main scheduling periodic task (HORLOGE) and 2 empty lists of functions defining the entry point of the attached periodic tasks (normal behavior and degraded behavior). At now, the HORLOGE task manages normal/degraded behavior mode switches; in future versions, this job will be performed by the OCERA FT controller.

The HORLOGE task performs undefinitely the following basic steps:

- wait for next period (using pthread_wait_np())
- check for faulty tasks (deadline misses)
- wake-up degraded behaviour tasks corresponding to the faulty tasks (this uses a synchronization semaphore; a barrier will be used instead in future versions) wake-up normal behaviour tasks for other tasks if needed (this uses a synchronization semaphore; a barrier will be used instead in future versions)

See also

_traiter_top_horloge, HORLOGE_ajouter_fct_periodique, HORLOGE_supprimer_fct_periodique

---

HORLOGE_terminer

Synopsis

```
void HORLOGE_terminer()
```

Include

"horloge.h"

Description

This function stops the main scheduling task (HORLOGE), all attached periodic tasks and frees allocated resources.

See also

HORLOGE_initialiser, HORLOGE_arreter, HORLOGE_supprimer_fct_periodique

---

HORLOGE_ajouter_fct_periodique

Synopsis

```
HorlogeId HORLOGE_ajouter_fct_periodique(const char *nom, HorlogeFct fct,
void *arg, double periode, uint_t priorite, size_t taillePile, HorlogeFct
fctDepassement, void *argDepassement)
```

Include

"horloge.h"

Parameters

nom

name of the periodic task to be created and attached

fct

normal behavior entry point function of the periodic task

arg

arguments to be passed to the normal behavior function

periode

period of the task (seconds)

priorite

priority of the task

taillePile

stack size of the task

fctDepassement

degraded behavior entry point function of the periodic task

argDepassement

arguments to be passed to the degraded behavior function

Return value

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

Description

This functions creates and attaches a task to the horloge component. At now, the function creates two RT tasks (one task for the normal behavior and one task for the degraded behavior) using standard pthread_create() to create both tasks, in future versions, the OCERA Fault-Tolerant component will be used instead to create the tasks pair using ft_task_create().

The period parameter (periode) should be a multiple of the HORLOGE period; if not, the period of the task is truncated to the lowest integer-multiple of the HORLOGE period.

The effective task priority (at RT-Linux level) is decreased by the priority of the main HORLOGE task (this ensures the attached periodic tasks have a lesser priority than the HORLOGE task; which is better;).

The normal behavior task performs undefinitely the following basic steps:

- wait for synchronization by the HORLOGE task (using a synch. sem)
- mark the task as active (static flag)
- call the normal behavior entry point function (fct)
- mark the task as inactive

The degraded behavior task performs undefinitely the following basic steps:

- wait for synchronization by the HORLOGE task (using a synch. sem)
- mark the task as active (static flag)
- call the degraded behavior entry point function (fctDepassement)
- mark the task as inactive

At now, normal/degraded mode switches are managed by HORLOGE task. In future versions, both tasks will synchronize on the same event (periodic timer) and the FT controller will do the job.

See also

HORLOGE_initialiser, HORLOGE_supprimer_fct_periodique

HORLOGE_activer_fct_periodique

Synopsis

```
int HORLOGE_activer_fct_periodique(HorlogeId id, int actif)
```

Include

"horloge.h"

Parameters

id

ID of the task to be activate/deactivate

actif

1 to activate, 0 to deactivate

Return value

always 0

Description

This function activates or deactivates calls to a periodic task. This does not perform a suspend or wake-up of the task but only activate or deactivate synchronization of the task by HORLOGE.

If this function is called when the task is running, the tasks finishes its job but will not be called anymore as long as it is activated again.

See also

HORLOGE_ajouter_fct_periodique

HORLOGE_supprimer_fct_periodique

Synopsis

```
int HORLOGE_supprimer_fct_periodique(HorlogeId id)
```

Include

"horloge.h"

Parameters

id

ID of the task to be detached

Return value

0 if successful, -1 otherwise

Description

The functions detaches the specified function from the HORLOGE managed tasks and cancels the corresponding tasks (normal and degraded behavior).

See also

HORLOGE_initialiser, HORLOGE_ajouter_fct_periodique

HORLOGE_forcer_execution

Synopsis

```
int HORLOGE_forcer_execution(HorlogeId id)
```

Include

"horloge.h"

Parameters

id

ID of the task

Description

This function starts immediately the specified task without waiting for the next period.

Return value

-1 on success, 0 otherwise

See also

HORLOGE_ajouter_fct_periodique

HORLOGE_arreter

Synopsis

```
void HORLOGE_arreter(void)
```

Include

"horloge.h"

Description

This function is not implemented yet.

See also

HORLOGE_redemarrer

HORLOGE_redemarrer

Synopsis

```
void HORLOGE_redemarrer(void)
```

Include

"horloge.h"

Description

This function is not implemented yet.

See also

HORLOGE_arreter

_point_d_entree

Synopsis

```
static void *_point_d_entree(void *arg)
```

Parameters

arg

ID of the task

Return value

Never returns (while (1))

Description

This function is the generic entry point for normal behavior of any periodic task. It calls the entry-point function associated to the normal behavior of the periodic task (this is the pointer 'fct' passed to function HORLOGE_ajouter_fct_periodique()).

_point_d_entree() performs undefinitely the following basic steps:

- wait for synchronization by the HORLOGE task (using a synch. sem)
- mark the task as active (static flag)
- call the normal behavior entry point function (fct)
- mark the task as inactive

See also

HORLOGE_ajouter_fct_periodique, _horloge_gerer_depassement

_horloge_gerer_depassement

Synopsis

```
static void *_horloge_gerer_depassement(void *arg)
```

Include

"horloge.h"

Parameters

arg

ignored

Return value

Never returns (while (1))

Description

This function is the generic entry point for degraded behavior of any periodic task. It calls the entry-point function associated to the degraded behavior of the periodic task (this is the pointer 'fctDepassement' passed to function HORLOGE_ajouter_fct_periodique()).

_horloge_gerer_depassement() performs undefinitely the following basic steps:

- wait for synchronization by the HORLOGE task (using a synch. sem)
- mark the task as active (static flag)
- call the degraded behavior entry point function (fctDepassement)
- mark the task as inactive

See also

HORLOGE_ajouter_fct_periodique, _horloge_gerer_depassement

_traiter_top_horloge

Synopsis

```
static void _traiter_top_horloge(int)
```

Description

This function is the entry point of the main HORLOGE task. It is a periodic task created using standard pthread_create() at application initialization by calling HORLOGE_initialiser() function.

HORLOGE acts as a scheduler on the attached periodic tasks; it is periodically woked-up by the RT-Linux task timer (using standard pthread_wait_np()) and performs undefinitely the following basic steps:

- wait for next period (pthread_wait_np())
- scan the list of attached periodic tasks
- check for faulty tasks (deadline miss at previous period)
- wake-up degraded behavior tasks corresponding to the tasks marked as faulty; this is done using a synchronization semaphore (but will be done with a barrier instead in future versions)
- wake-up normal behavior tasks corresponding to the tasks not marked as faulty; this is done using a synchronization semaphore (but will be done with a barrier instead in future versions)

At now, as described above, _traiter_top_horloge() self manages deadline misses on attached

periodic tasks. In future versions of the component, this will be performed using the OCERA Fault-Tolerant components instead.

See also

HORLOGE_initialiser, HORLOGE_ajouter_fct_periodique

3.4 Implementation issues

Not applicable

3.5 Tests and validation

Unit test application provided with component and Robotic application validation.

3.5.1 Validation criteria

Unit test application success and Robotic application behavior as expected.

3.5.2 Tests

See unit test application.

3.5.3 Results and comments

Component successfully tested.

3.6 Examples

None

3.7 Installation instructions

Not applicable

Chapter 4. protocol component

4.1 Summary

Name:

protocol

Description:

Provide simple protocol for communication between the Controller application and the Client application.

Author:

F. Russotto (russotto@cea.fr)

Reviewer:

Layer:

Application level

Version:

0.1 beta

Status:

Design

Dependencies:

rtl, rtl_sched, rtl_fifo, rtl_malloc, generic robotics component.

Release date:

MS 4

4.2 Description

This component provides a simple protocol for communication between the Controller application and the Client application. The component uses standard RT fifos to exchange data with a user-space process (see proto_interface) which exchange data with the Client application using standard Linux sockets. The communication protocol is based on UDP/IP.

4.3 API / compatibility

`server_read`
Receive Controller input data from the Client application

`server_write`
Send Controller output data to the Client Application

do_proto
: Synchronise Controller / Client data exchange

init_proto
: Initialize the protocol component

uninit_proto
: Uninitialize the protocol component

write_protocol
: Compute a checksum and write raw data

read_protocol
: Read raw data and verify the checksum

_fifo_server_handler
: Client -> Controller communication fifo handler

write_udp
: Send raw data to the Client application

read_udp
: Receive raw data from the Client application

init_module_udp
: Initialize the proto_udp sub-component (protocol)

cleanup_module_udp
: Cleanup the proto_udp sub-component (protocol)

---

server_read

Synopsis

```
static void * server_read(void)
```

Include

"protocol.h"

Return value

never returns (while (1))

Description

This function is the entry point of the PROTO_IN task. PROTO_IN is an asynchronous sporadic task providing the following service:

- receive control data coming from the Client application through an Ethernet connection
- decode data from the Sun XDR format to the local platform format

Incoming data is preliminary received by a user-space process (part of the proto_interface component) on a standard Linux socket, using UDP/IP protocol. Data is then transfered to the PROTO_IN task through a dedicated RT-fifo using read_protocol(). server_read() performs undefinitely the following steps:

- receive checked data (using read_protocol())
- lock the _mtx_proto mutex (TR_attendre_semaphore())
- decode data and store it to a shared static structure VPr (encode_decode())
- unlock the _mtx_proto mutex (TR_poster_semaphore())

read_protocol() is a blocking function; it does not return until a full checked data is available for decoding. As soon as data is available, it is immediately decoded and stored in the VPr static structure for use by the do_proto() function. The _mtx_proto mutex prevents data from being accessed in do_proto() during storage. Using such a dedicated task (PROTO_IN) and such a mechanism allows to fully de-synchronize the Soft Real-Time context where Ethernet data comes from, from the Hard Real-Time context where the SERVO task takes place. server_read() is declared static as it is private to the protocol component. The PROTO_IN task is started in the init_proto() function see below) which should be called at application initialization.

See also

read_protocol, server_write, do_proto, init_proto

---

server_write

Synopsis

```
static void server_write(void)
```

Include

"protocol.h"

Description

This function provides the following service to the application:

- encode state data from the local platform format to the Sun XDR format
- send data to the Client application through an Ethernet connection

Outgoing data is actually sent to a user-space process (part of the proto_interface user-space component) through a dedicated RT-fifo. Data is then transmitted by the user-space process to the Client application through a standard Linux socket using UDP/IP. server_write() performs the following steps:

- encode data from shared static structure VP (encode_decode())
- send encoded data (write_protocol())

server_write() is declared static as it is private to the protocol component; it is called within the do_proto() function (see below).

See also

do_proto, server_read

---

do_proto

Synopsis

```
int do_proto(ServoDonnees * don)
```

Include

"protocol.h"

Parameters

don

pointer to the main application data structure

Return value

always 0

Description

This function performs the 3 following basic steps:

- copy control data from the global structure don to the shared static structure VPr (copie_consigne())
- copy state data from the shared static structure VP to the global global structure don (copie_etat())
- call server_write()

Parameter don is a pointer to the main data structure of the application; this data structure contains all control and state variables needed at the different steps of the main servo-control loop performed by the SERVO task (for details on the main servo- control loop and the SERVO task, see hapticctrl component). do_proto() is called within the synchronous hard-Real-Time context of the SERVO task.

See also

copie_consigne, copie_etat, server_write

---

init_proto

Synopsis

```
int init_proto(void)
```

Include

"protocol.h"

Return value

0 if succesfull, -1 otherwise.

Description

This function performs the 4 following steps:

- initialize static semaphore _sem_fifo_server to TAKEN (sem_init())
- initialize some component internal parameters
- create static mutex _mtx_proto
- start the PROTO_IN task (entry point: server_read())

_sem_fifo_server is a synchronization semaphore used to provide blocking behavior to the read_protocol() function (see read_protocol). _mtx_proto is a mutex semaphore used to protect concurrent accesses to the VPr static structure, as it is accessed by both the SERVO task (do_proto()/copie_consigne()) and the PROTO_IN task (server_read()). init_proto() should be called at application initialization by the INIT task (thought this has not been tested, it **might** be called in the application init_module() function; in such a case, init_module_udp() should be called first).

See also

init_module_udp, do_proto, server_read, uninit_proto

---

uninit_proto

Synopsis

```
int uninit_proto(void)
```

Include

"protocol.h"

Return value

0 if succesfull, -1 otherwise.

Description

This function performs the 4 following steps:

- cancel the PROTO_IN task
- delete mutex _mtx_proto
- unitialize some component internal parameters

uninit_proto() should be called at application termination (eg: in the application cleanup_module() function); the cleanup_module_udp() function should be called after call to uninit_proto().

See also

cleanup_module_udp, init_proto

---

write_protocol

Synopsis

```
ssize_t write_protocol(int fildes, const void *buf, size_t nbyte)
```

Include

"protocol.h"

Parameters

fildes

file descriptor (not used under RTLinux)

buf

pointer to data to be sent

nbyte

data size in bytes

Return value

nbyte if successful, -1 otherwise

Description

This function is called within the server_write() function. It computes a checksum for outgoing data and reformat data as a raw packet including a header, the data size, the data and the computed checksum. Reformated raw data is then passed to the proto_write() function.

See also

proto_write, read_protocol

---

read_protocol

Synopsis

```
ssize_t read_protocol(int fildes, const void *buf, size_t nbyte)
```

Include

"protocol.h"

Parameters

fildes

file descriptor (not used under RTLinux)

buf

pointer to data to be read

nbyte

maximum data size in bytes

Return value

read data size in bytes if successful, -1 otherwise

Description

This function is called within the server_read() function and performs the following steps:

- retrieve from the Client application a raw data packet consisting of: a header, a data size, a data, and a checksum (function proto_read())
- extract data, data size and checksum from the packet
- compute a new checksum from the extracted data
- compare computed and extracted checksums
- reformat data so that it is usable by server_read() (removes header data size and checksum).

read_protocol() is a blocking function; it does not return until a full checked data is available or an error occurred.

See also

proto_write

_fifo_server_handler

Synopsis

```
static int _fifo_server_handler(unsigned int fifo)
```

Include

"protocol.h"

Parameters

fifo

RT-fifo (device) number

Return value

not applicable

Description

This function is the handler associated to the Client -> Controller communication fifo (FIFO_SERVER). A fifo handler is automatically called when a data is available in the fifo. This behavior allows synchronization between user-space and kernel-space using a simple synchronization mechanism such as a synchronization semaphore. The only job carried out by _fifo_server_handler() is to post the _sem_fifo_server synchronization semaphore (using sem_post()). Doing so wakes up the PROTO_IN task which is waiting for synchronization within the read_udp() function.

See also

read_udp

---

write_udp

Synopsis

```
static ssize_t write_udp (int fildes, void *buff, size_t len)
```

Include

"protocol.h"

Parameters

fildes

file descriptor (not used under RTLinux)

buf

pointer to data to be sent

nbyte

data size in bytes

Return value

len if successful, -1 otherwise

Description

This function sends raw data packet generated by write_protocol() to the Client application. Outgoing raw data is actually sent by a user-space process (see proto_interface component) using a standard Linux socket; data is read by the user-space process from a dedicated RT fifo. write_udp() writes this data to the fifo. write_udp() performs the following steps:

- serialize data length and data into a unique buffer
- write the buffer to the fifo

write_udp() is called within the write_protocol() function.

See also

write_protocol, read_udp

---

read_udp

Synopsis

```
static ssize_t read_udp (int fildes, void *buff, size_t len)
```

Include

"protocol.h"

Parameters

fildes

file descriptor (not used under RTLinux)

buf

pointer to data to be read

nbyte

maximum data size in bytes

Return value

read data size in bytes if successful, -1 otherwise

Description

This function receives raw data sent by the Client application to the Controller application. Incoming raw data is actually received by a user-space process (see proto_interface component) using a standard Linux socket; it is then sent to the kernel-module application using a RT-fifo. A handler associated to the fifo (_fifo_server_handler()) and a synchronization semaphore (_sem_fifo_server) allows read_udp() to provide a blocking behavior. read_udp() performs the following steps:

- wait for synchronization (sem_wait(_sem_fifo_server))
- retrieve data from the fifo (rtf_get())
- if data length is complete, then return from call else, loop until data length is complete

Note that the _sem_fifo_server synchronization semaphore is not posted back by read_udp(); it is actually posted by the fifo handler _fifo_server_handler(). This ensures the blocking behavior of read_udp(). read_udp() is called within the read_protocol() function.

See also

_fifo_server_handler, write_udp, read_protocol

---

init_module_udp

Synopsis

```
int init_module_udp(int fifo_client, int fifo_server)
```

Include

"protocol.h"

Parameters

fifo_client

RT-fifo to be used for

Controller

Client communication

fifo_server

RT-fifo to be used for

Client

Controller communication

Return value

0 if succesfull, -1 otherwise.

Description

This function creates the two RT-fifos needed by the protocol component to communicate with the user-space proto_interface component (FIFO_SERVER & FIFO_CLIENT). It also creates a handler associated to the FIFO_SERVER fifo using the static function: _fifo_server_handler(); this handler is used to provide a blocking behavior to the read_protocol() function. This function should be called within the application init_module() function and prior any call to init_proto().

See also

init_proto, cleanup_module_udp

---

cleanup_module_udp

Synopsis

```
int cleanup_module_udp(void)
```

Include

"protocol.h"

Return value

0 if succesfull, -1 otherwise.

Description

This function destroys previously created FIFO_SERVER & FIFO_CLIENT RT fifos. This function should be called within the application cleanup_module() function and only after call to uninit_proto().

See also

init_proto, cleanup_module_udp

4.4 Implementation issues

Not applicable

4.5 Tests and validation

Robotic application validation.

4.5.1 Validation criteria

Robotic application behavior as expected.

4.5.2 Tests

Robotic application.

4.5.3 Results and comments

Robotic application works as expected.

4.6 Examples

None

4.7 Installation instructions

Not applicable

Chapter 5. hapticctrl component

5.1 Summary

Name:

hapticctrl

Description:

Robotic application kernel-module top-level component.

Author:

F. Russotto (russotto@cea.fr)

Reviewer:

Layer:

Application level

Version:

0.1 beta

Status:

Design

Dependencies:

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

Release date:

MS 4

5.2 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 & functions (eg: init_module, cleanup_module, …)
- the main application initialization function (hapticctrl)
- Virtuose robot arm simulator

5.3 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)

ES_standard_build_logical_state
Update System state

ES_standard_send_commands
Send motor commands to the simulated Virtuose robot

_sim_virtuose_dynamics
Simulate a Virtuose robot arm

_sim_get_position
Get robot handle position from the Controller internal model

_sim_set_force
Set user force/torque on handle of the Controller internal model

_sim_ctrl_fifo_handler
Controller mode change fifo handler

_sim_force_fifo_handler
User force/torque set fifo handler

_sim_send_position
Send simulated Virtuose robot arm handle position to the virtsim HMI

---

init_module

Synopsis

```
int init_module(void)
```

Return value

Always 0

Description

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

- Create RT-fifos used by the application; this includes 2 fifos for protocol <-> proto_interface (user-space) communication, 3 fifos for Virtuose simulator <-> virtsim (user-space) communication and 5 general-purpose RT-fifos used by the temps-reel component
- Create fifo handlers for all user-space -> kernel space fifos
- Create the main application initialization task (INIT) using standard pthread_create() function of the RTLinux API.

Most of application initializations are made in the hapticctrl() function; the init_module() function is only used to create objects that RTLinux does not allow to create in a RT context (such as fifos). We all expect that RTLinux will allow RT-fifo creation in a RT context in the future. The INIT task, created in the init_module() function 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, destroys created RT-fifos 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 initalization task (INIT). INIT task is created in the application init_module() after creation of needed RT-fifos and fifo handlers. The hapticctrl() function (mainly) performs the following steps:

- initialize required haptics & robotics components
- initialize protocol component (init_proto(); this will create the PROTO_IN task)
- initialize horloge component (HORLOGE_initialiser())
- create the main SERVO task (HORLOGE_ajouter_fct_periodique(); this will create the HORLOGE and DLMISS tasks)

See also

init_proto, HORLOGE_initialiser, HORLOGE_ajouter_fct_periodique, _servo

---

_servo

Synopsis

```
void _servo(void)
```

Description

This function is the entry point of the main servo-control task (SERVO)

See also

hapticctrl

ES_standard_build_logical_state

Synopsis

```
void ES_standard_build_logical_state(ServoDonnees * don, ulong_t estampille)
```

Parameters

don

pointer to the main application data structure

estampille

period number (horloge component reference)

Description

This function is called within the main synchronous servo-control loop (see hapticctrl). It is (normally) used to update logical state of the System (eg: the set of logical variables that define the System mode); the System considered here is the Controller + the controlled Robot. In the context of OCERA robotic application V1, this function is also used to:

- get position of the simulated Virtuose robot (using call to _sim_get_position())
- set force/torque to be applied to the simulated Virtuose
- send position to the Virtuose HMI (_sim_send_position())

The Virtuose HMI is a user-space process (see virtsim component) implementing a Human-Machine Interface allowing user to control and monitor the simulated Virtuose robot (eg: position monitoring, and force/torque control). Communication between virtsim and the Controller application kernel-module uses 3 dedicated RT-fifos (1 output, 2 input). Function _sim_send_position() uses the output fifo to send position data to virtsim synchronously (eg: within the hard RT context of the SERVO task); data can be sent each period (~ 1 ms) or each N periods if needed. Force/torque data coming from virtsim are collected asynchronously using a handler associated to input fifo #1 (see _sim_force_fifo_handler()) while System mode change control words coming from virtsim are catched by a handler associated to input fifo #2 (see _sim_ctrl_fifo_handler()).

See also

_sim_get_position, _sim_set_force, _sim_send_position, _sim_force_fifo_handler, _sim_ctrl_fifo_handler

---

ES_standard_send_commands

Synopsis

```
void ES_standard_send_commands(ServoDonnees * don)
```

Parameters

don

pointer to the main application data structure

Description

This function is called within the main synchronous servo-control loop (see hapticctrl). It is (normally) used to send computed commands to the motors of the controlled robot. In the context of OCERA robotic application V1, instead of sending commands to a real hardware card connected to a true robot, commands are passed to a simulation function (_sim_virtuose_dynamics()) that computes dynamic behavior of a virtual Virtuose robot arm. As mentioned above, dynamic computation is done during the synchronous hard- RT context of the servo-control loop; this introduces a minor overhead in the servo-control loop but not so much as the simulation model is very simplistic.

See also

_sim_virtuose_dynamics

_sim_virtuose_dynamics

Synopsis

```
static void _sim_virtuose_dynamics(ServoDonnees * don)
```

Parameters

don

pointer to the main application data structure

Description

This function is called within the main synchronous servo-control loop (see hapticctrl) by the ES_standard_send_commands() function. _sim_virtuose_dynamics() simulates (computes) dynamic behavior of a virtual Virtuose robot arm. The robot model used is very simplistic to avoid introducing non-representative overheads within the servo-control loop; this involves a dynamic behavior of the simulated robot which is not as close as possible to the real one but this does not matter in any way in the OCERA context as we are not testing the Controller application itself but the OCERA system components;). Dynamic robot simulation inputs are the following:

- motor commands (data computed by the Controller)
- force/torque applied by the user on the Virtuose handle (data coming from the virtsim HMI; see ES_standard_build_logical_state())

While simulation output are the following:

- position of the Virtuose handle (data sent to the virtsim HMI)

As _sim_virtuose_dynamics() computes the robot behavior in the motors reference, we need to express:

- force/torque applied on the handle in the motors reference
- handle position in any reference usable by the virtsim HMI such as: cartesian (base) reference or articular reference

Still to avoid non representative overheads, _sim_virtuose_dynamics() does not compute these transformations. Instead of that we use the Controller internal model of the robot to:

- apply force/torque on the Controller internal model as a disturbing force/torque right before the motors commands are computed (this is done in the _sim_set_force() function)
- retrieve handle position of the Controller internal model in cartesian (base) and articular reference (this is done in the _sim_get_position() function

By this way, the computed pseudo-motor commands retrieved and used in _sim_virtuose_dynamics() are the sum of the effective motor commands (as computed in the control loop) and the user force/torque applied on handle in the motors reference (take it easy;).

See also

ES_standard_send_commands, _sim_get_position, _sim_set_force

_sim_get_position

Synopsis

```
static void _sim_get_position(ServoDonnees * don)
```

Parameters

don

pointer to the main application data structure

Description

This function is called within the main synchronous servo-control loop (see hapticctrl) by the ES_standard_build_logical_state() function. _sim_get_position() retrieve handle position of the Controller internal robot model expressed in both the cartesian (base) and the articular references. Data is copied to a static structure (virtuose) for use by the _sim_send_position() function.

See also

_sim_send_position, _sim_set_force, ES_standard_build_logical_state

---

_sim_set_force

Synopsis

```
static void _sim_set_force(ServoDonnees * don)
```

Parameters

don

pointer to the main application data structure

Description

This function is called within the main synchronous servo-control loop (see hapticctrl) by the ES_standard_build_logical_state() function. _sim_set_force() sets a disturbing force/torque on the handle of the Controller internal model; this disturbing force/torque corresponds to a user force/torque applied on the simulated Virtuose robot originated from the virtsim HMI (see virtsim component). Force/torque data used as input is stored in a static array (_sim_p_force) that is asynchronously refreshed by the RT-fifo handler: _sim_force_fifo_handler() as soon as a new data coming from the virtsim HMI is available in the fifo.

See also

_sim_get_position, _sim_force_fifo_handler, ES_standard_build_logical_state

---

_sim_ctrl_fifo_handler

Synopsis

```
static int _sim_ctrl_fifo_handler(unsigned int fifo)
```

Parameters

fifo

RT-fifo (device) number

Description

This function is the handler associated to the FIFO_SIM_CTRL RT-fifo. This RT-fifo is used by the virtsim HMI user-space process to send System mode change control words to the application kernel-module. This allows to power on or off Controller, to request power state, to enable/disable HMI/application communication or to trigger an emergency stop. Incoming data is atomic (int), so there is no need to ensure data consistency using rtf_get().

See also

_sim_force_fifo_handler

_sim_force_fifo_handler

Synopsis

```
static int_sim_force_fifo_handler(unsigned int fifo)
```

Parameters

fifo

RT-fifo (device) number

Description

This function is the handler associated to the FIFO_SIM_FORCE RT-fifo. This RT-fifo is used by the virtsim HMI user-space process to send force/torque applied by user on the simulated Virtuose robot arm handle. Force/torque data is an array of 6 floats. A simple mechanism using a temporary array prevents data inconsistency: rtf_get'ed data is preliminary copied to the temporary array; it is then flushed to the shared static array _sim_p_force only when the 6 floats of the array have been fully rtf_get'ed.

See also

_sim_set_force, _sim_ctrl_fifo_handler

_sim_send_position

Synopsis

```
static int _sim_send_position(void)
```

Return value

number of bytes sent

Description

This function is called within the main synchronous servo-control loop (see hapticctrl) by the ES_standard_send_commands() function. _sim_send_position() sends to virtsim HMI user-space process the computed position of the simulated Virtuose robot arm using a RT-fifo (FIFO_SIM_POSITION). The rtf_put'ed data is stored in a shared static structure (_sim_position) computed by the _sim_get_position() function.

See also

_sim_set_position, ES_standard_send_commands

5.4 Implementation issues

Not applicable

5.5 Tests and validation

Robotic application validation.

5.5.1 Validation criteria

Robotic application behavior as expected.

5.5.2 Tests

Robotic application execution.

5.5.3 Results and comments

Robotic application works as expected.

5.6 Examples

None

5.7 Installation instructions

Not applicable

Chapter 6. proto_interface component

6.1 Summary

Name:

proto_interface

Description:

Provide user-space side of the protocol component using Linux sockets

Author:

F. Russotto (russotto@cea.fr)

Reviewer:

Layer:

Application level

Version:

0.1 beta

Status:

Design

Dependencies:

libc, socket, pthread, rtl_fifo.

Release date:

MS 4

6.2 Description

This user-space component is connected to the protocol kernel component. It provides user-space side of the communication protocol (see protocol component), allowing access to standard Linux sockets for communication between the Client application and the Controller application.

6.3 API / compatibility

`_server`
SERVER user-space thread entry point

`_client`
CLIENT user-space thread entry point

`main`
Communication protocol interface process entry point

_server

Synopsis

```
void _server(void)
```

Description

This function is the entry point of the SERVER thread. SERVER is a user-space posix thread which undefinitely listens to a Linux socket for incoming data transmitted by the Client application and sends it to the Controller kernel-module application through a RT-fifo. _server() performs undefinitely the following steps:

- receive incoming data from an open socket (UDP/IP)
- send received data to the Controller kernel-module application using a RT-fifo (FIFO_SERVER)

See also

_client, main

_client

Synopsis

```
void _client(void)
```

Description

This function is the entry point of the CLIENT thread. CLIENT is a user-space posix thread which undefinitely reads outgoing data from a dedicated RT-fifo and sends it to the Client application through a standard Linux socket using UDP/IP. _server() performs the following steps:

- read outgoing data from a RT-fifo (FIFO_CLIENT)
- send data to the Client application through a Linux socket.

See also

_server, main

---

main

Synopsis

```
int main(void)
```

Return value

0 on execution success, non-zero otherwise

Description

This function is the entry point of the Communication protocol user-space interface (PROTO_INTERFACE). The tasks carried out main() are the following:

- open and bind a UDP socket for the SERVER & CLIENT threads
- open FIFO_SERVER RT-fifo for writing
- open FIFO_CLIENT RT-fifo for reading
- initialize CLIENT & SERVER posix threads attributes to: priority=99 (max), scheduling=SCHED_FIFO, ...
- create CLIENT & SERVER posix threads (pthread_create())
- wait for threads termination (pthread_join())
- close socket, RT-fifos and exit

See also

_server, _client

6.4 Implementation issues

Not applicable

6.5 Tests and validation

Robotic application validation.

6.5.1 Validation criteria

Robotic application behavior as expected.

6.5.2 Tests

Robotic application.

6.5.3 Results and comments

Robotic application works as expected.

6.6 Examples

None

6.7 Installation instructions

Not applicable

Chapter 7. virtsim component

7.1 Summary

Name:

virtsim

Description:

Virtuose robot arm simulator user interface.

Author:

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

Reviewer:

Layer:

Application level

Version:

0.1 beta

Status:

Design

Dependencies:

libc, pthread, rtl_fifo.

Release date:

MS 4

7.2 Description

This component implements the Virtuose robot arm simulator HMI. Virtuose HMI (virtsim) is a user-space process that allows user to control and monitor the simulated Virtuose robot (eg: position monitoring, and force/torque control).

virtsim component has 2 two different implementations:

- Implementation #1 is a very simple full-text based interface. It has been developed to debug and test the overall application behaviour.
- Implementation #2 is a QT-based Graphical User Interface much more sexy than the 1st one. Unfortunately this implementation could not be tested successfully as it introduced severe instabilities in RT-Linux latencies (tons of deadline misses each second, latencies up to 100 milliseconds); tests proved that virtsim was not the cause itself as there was absolutely no link (communication) between virtsim and the Linux kernel during tests. The QT implementation is given as is and will not be linked to the Controller application until a Linux / RT-Linux bug fix is available.

7.3 API / compatibility

`main(#1)`
Virtuose robot arm simulator text-based HMI entry point (virtsim #1)

`_hmi`
HMI posix thread entry point

`main(#2)`
Virtuose robot arm simulator QT-based HMI entry point (virtsim #2)

`VirtSimDlg::timerEvent`
QT Timer event callback method

`VirtWidget::paintGL()`
Glut paint method

---

main(#1)

Synopsis

```
int main(void)
```

Return value

0 on execution success, non-zero otherwise

Description

This function is the entry point of the Virtuose robot arm simulator user-space text-based HMI (virtsim #1).

Communication between virtsim and the Controller application kernel-module uses 3 dedicated RT-fifos (1 input, 2 output). The input fifo is used to read position data coming from the Controller application kernel-module while output fifo #1 is used to send user force/torque data and fifo #2 to send System mode change control words.

Data exchange between virtsim and the Controller application kernel module are made in a dedicated posix thread (HMI) that main() creates, while main() is used for user inputs. The tasks carried out are:

- open 3 RT-fifos (1 input, 2 output) to communicate with kernel
- set attributes and create the HMI posix thread (priority=97, scheduling=SCHED_FIFO)
- loop undefinitely waiting for user inputs; available user imputs are: power on/off simulated Virtuose robot, apply force/torque on Virtuose robot handle, exit virtsim; user force/torque data is sotred in a shared static array (_sim_force)
- exit loop on user request
- order HMI thread to terminate
- wait for HMI thread termination (pthread_join())
- close RT-fifos
- exit process

See also

_hmi

_hmi

Synopsis

```
void _hmi(void)
```

Description

This function is the entry point of the HMI thread. HMI thread is created by the main process to communicate both ways with the Controller application kernel-module and optionnally displays simulated Virtuose robot position on screen. _hmi() performs the following tasks:

- open HMI / Controller app communication by sending an appropriate control word to fifo #2
- do undefinitely:
- read position data from input fifo (blocking), then store it in a shared static structure (_sim_position)
- write force/torque data from a shared static structure (_sim_force) to output fifo #1
- (optionally) displays position on screen
- exit loop on user request
- close HMI / Controller communication
- exit thread

See also

main

---

main(#2)

Synopsis

```
int main(void)
```

Return value

0 on execution success, non-zero otherwise

Description

This function is the entry point of the Virtuose robot arm simulator user-space QT-based graphical HMI (virtsim #2).

Communication between virtsim and the Controller application kernel-module uses 3 dedicated RT-fifos (1 input, 2 output). The input fifo is used to read position data coming from the Controller application kernel-module while output fifo #1 is used to send user force/torque data and fifo #2 to send System mode change control words. Data exchange between virtsim and the Controller application kernel module are made in a dedicated posix thread (HMI) that main() creates, while main() is mainly used to create the QT GUI. The main() function performs the following tasks:

- create the 3 needed RT-fifos to communicate with the Controller application kernel-module
- set attributes and create the HMI posix thread (priority=97, scheduling=SCHED_FIFO)
- create and show the main dialog box object (class VirtDlg); VirtDlg constructor will (among other things) create the Glut widget object (class VirtWidget) which is used to display a 3D model of the Virtuose robot arm
- destroys RT-fifos exit process

See also

_hmi, VirtSimDlg::timerEvent

VirtSimDlg::timerEvent

Synopsis

```
void VirtSimDlg::timerEvent( QTimerEvent * )
```

Description

This function implements the QT Timer event call-back method. It is periodically called by the low-level QT layer at a specified rate (12 to 50 Hz). VirtSimDlg::timerEvent() performs the following tasks:

- check for key pressed by user
- set force/torque to be applied on Virtuose robot handle depending on key pressed by user then store force/torque data in the shared static array _sim_force
- call the VirtWidget::updateGL() that will (among other things) call the VirtWidget::paintGL() method

See also

VirtWidget::paintGL

VirtWidget::paintGL()

Synopsis

```
void VirtWidget::paintGL()
```

Description

This function implements the painting method of the VirtWidget class. This method is called periodically by the VirtWidget::updateGL() method (auto-generated by QT designer) which is called within the VirtSimDlg::timerEvent() method.

paintGL() redraws the Virtuose robot 3D model into the application dialog box, using Virtuose robot position parameters stored in the shared static structure _sim_position.

See also

VirtSimDlg::timerEvent

7.4 Implementation issues

Not applicable

7.5 Tests and validation

Robotic application validation.

7.5.1 Validation criteria

Robotic application behavior as expected.

7.5.2 Tests

Robotic application.

7.5.3 Results and comments

Robotic application works as expected.

7.6 Examples

None

7.7 Installation instructions

Not applicable

Chapter 8. Conclusion and future works

OCERA Robotic application V1 is still in a very beta state but provides the expected behavior. The few remaining bugs will be fixed soon and some internal mechanisms of the application will be completed or modified in order to enhance application efficiency using the following OCERA components :

- Fault-Tolerance component in temps_reel and horloge components
- POSIX barriers instead of synchronization semaphore in horloge
- POSIX Trace in temps_reel_debug component

Future works on the application leading to version V2 will mainly be focused on:

- minor architectural changes of the application in order to fit the new industrial version (some changes to enhance application efficiency in the Client/Controller communication mechanisms, and in some robotics/haptics mechanisms)
- use of RTL-lwIP component (if available in OCERA) to avoid use of a Linux soft-real time interface for Client/Controller communication through UDP
- application integration on a real (hardware) Virtuose Robot arm

Chapter 9. Acronyms table

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	Degrees of Freedom
FIFO	First In First Out (ITC mechanism)
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
UDP	User Datagram Protocol
XDR	eXternal Data Representation (Sun microsystems)