Temps réel Linux et Xenomai

Approches du temps réel strict avec Linux

Christophe Blaess

christophe.blaess@logilin.fr

https://www.blaess.fr/christophe/
twitter: @chrisblaess

Ingénierie et formations sur Linux et les logiciels libres
https://www.logilin.fr

Approches du temps-réel strict avec Linux.....3
Noyau standard.....3
RT-Linux dans les années 1990.....4
Début des années 2000 : Adeos.....5
RTAI en mode noyau.....6
RTAI en mode utilisateur : LxRT.....7
Xenomai.....8
Installation et tests de Xenomai.....9
Test de Xenomai.....12
Programmation sous Xenomai.....13
Modes primaire et secondaire.....14
Gestion des tâches.....15
Tâches périodiques.....16
Sommeil et attentes.....16
Mutex et sémaphores.....17
Messages synchrones.....18
Files de messages.....18

Ce cours a été rédigé en utilisant des logiciels libres sur système d'exploitation Linux :

- *LibreOffice Writer* pour le support et la mise en page
- *LibreOffice Draw* pour les dessins vectoriels
- *Gimp* pour les images bitmap

Temps réel Linux et Xenomai

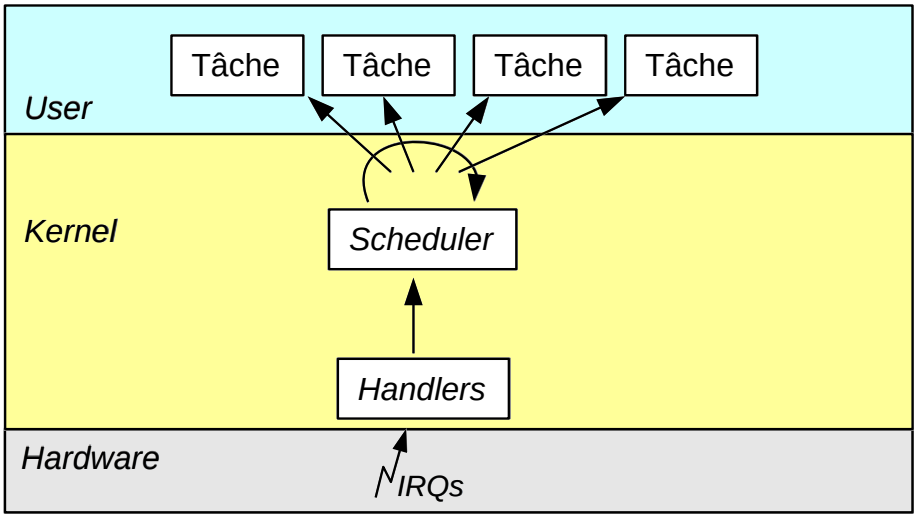
v. 5.3

https://www.blaess.fr/christophe/

https://www.logilin.fr

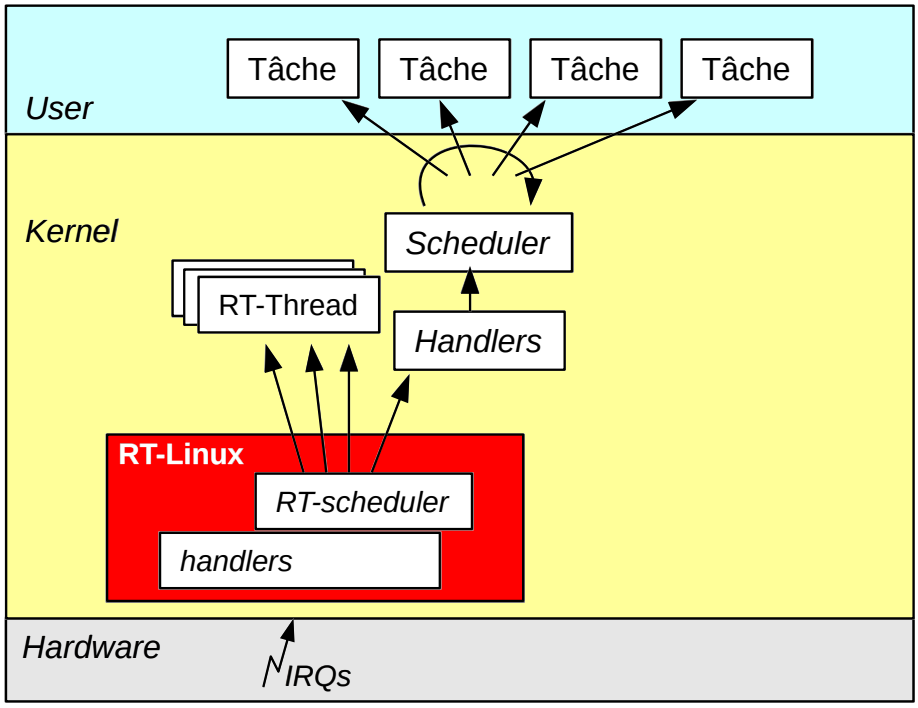
Approches du temps-réel strict avec Linux

Noyau standard



Dans le noyau Linux standard, les interruptions déclenchées par les périphériques (ou les exceptions provenant du processeur) sont gérées par des *handlers*. Au retour d'un *handler*, l'ordonnanceur du noyau est invoqué. Ce dernier active les tâches de l'espace utilisateur en fonction de leurs priorités temps-partagé ou temps-réel souple.

RT-Linux dans les années 1990

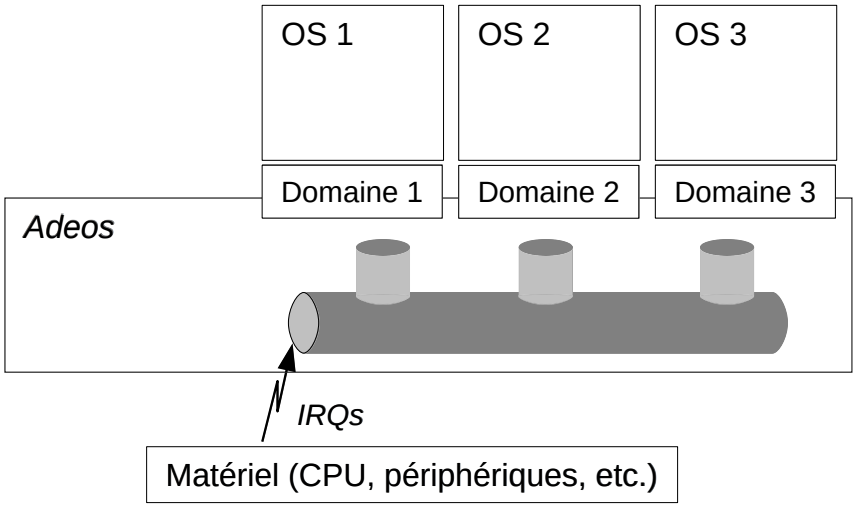


Le système RT-Linux, créé par Victor Yodaiken dans les années 1990, puis commercialisé par FSM-labs (puis Wind River) ajoute un « *nano-kernel* » qui reçoit les interruptions avant le noyau Linux, et intègre un petit ordonnanceur minimal permettant d'activer des tâches temps-réel strictes plus prioritaires que le noyau Linux.

Dans la première version de RT-Linux, les tâches temps-réel étaient implémentées sous forme de threads s'exécutant dans l'espace mémoire du noyau Linux. Ils pouvaient communiquer avec les processus de l'espace utilisateur par des pages de mémoire partagée et par des Fifos temps-réel.

FSM-labs a breveté ce concept, empêchant le développement du projet libre RTAI qui était basé sur le même principe.

Début des années 2000 : Adeos



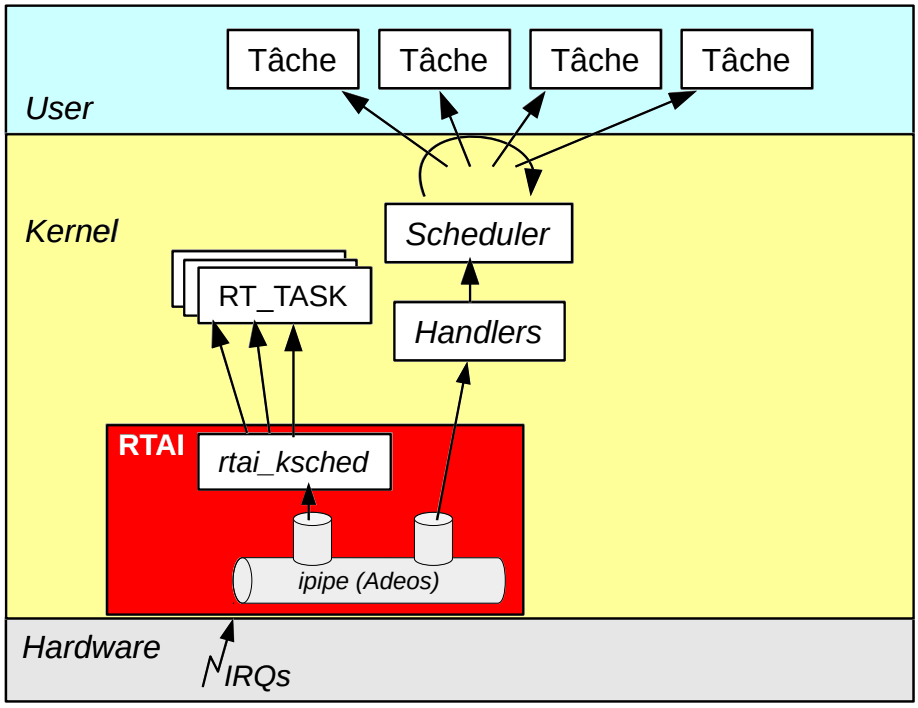
Adeos est une couche de virtualisation qui capture les interruptions matérielles et les distribue par l'intermédiaire d'un pipeline (*interrupt pipeline*) aux différents systèmes d'exploitation supportés.

Décrit par l'article de Karim Yaghmour, et contournant les termes du brevet de FSM-labs, ce système fut implémenté par Philippe Gerum en 2002.

Voir aussi

[Yaghmour 2001] : Karim Yaghmour « *Adaptative Domain Environment for Operating Systems* »

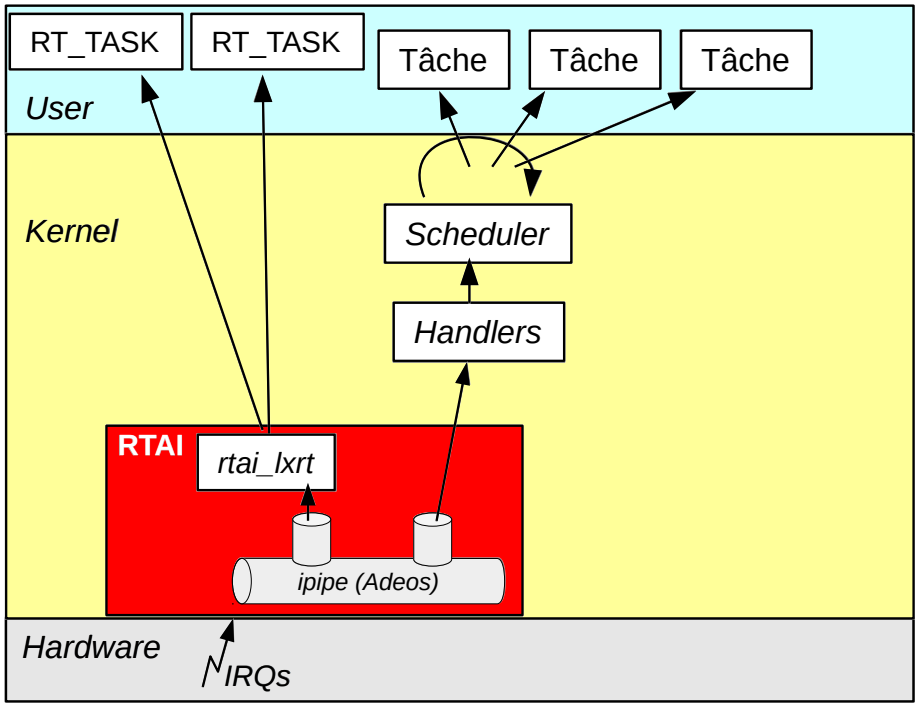
RTAI en mode noyau



RTAI utilise Adeos (renommé *ipipe*) pour virtualiser les interruptions. L'ordonnanceur temps-réel *rtai_ksched* reçoit les interruptions en premier et actives des tâches RTAI. Une fois tous les traitements de *rtai_ksched* terminés, les interruptions sont transmises aux *handlers* du noyau Linux.

Les tâches temps-réel de RTAI et les processus du mode utilisateur Linux peuvent dialoguer par l'intermédiaire de FIFO temps-réel (module *rtai_fifos*), de segments de mémoire partagée (*rtai_shm*), etc.

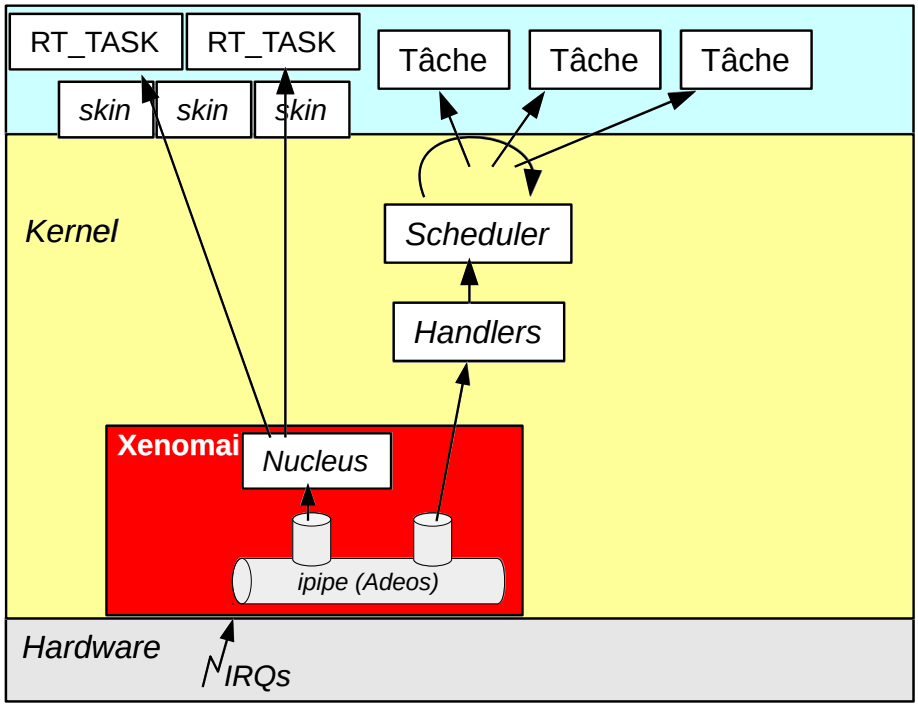
RTAI en mode utilisateur : LxRT



Avec le module LxRT (*Linux Real-Time*) de RTAI, il est possible d'ordonnancer temporairement un processus de l'espace utilisateur avec une priorité temps-réel stricte.

Durant ce temps, le processus ne doit pas faire appel aux services du noyau Linux (appels-système).

Xenomai



Xenomai est écrit conjointement à Adeos, il fonctionne sur un mode assez proche de RTAI/LxRT.

Xenomai propose une API native comportant toutes les fonctionnalités pour faire fonctionner un système temps-réel :

- création, destruction de tâches, suspension et redémarrage, terminaison et attente d'une autre tâche ;
- basculement d'une tâche en mode périodique, mise en sommeil, attente d'un timer ;
- gestion des interruptions, installation d'un handler temps-réel ;
- sémaphores, mutex, variables-conditions, tubes et files de messages...

Cette API est suffisamment générale pour être recouverte par des *skins* émulant des API classiques :

- Posix PSE51 (pthreads, timers, mutex, sémaphores, mémoire partagée, interruptions...) ;
- pSos+ ;
- RTAI
- Vrtx
- VxWorks

Installation et tests de Xenomai

Choisir un patch ipipe correspondant à l'architecture et la branche du noyau voulues :

```
$ wget  https://www.xenomai.org/downloads/ipipe/v5.x/x86/ipipe-core-5.4.228-x86-12.patch
```

Télécharger et extraire le noyau correspondant au patch :

```
$ wget  https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.4.228.tar.xz
$ tar  xf  linux-5.4.228.tar.xz
```

Cloner les sources de Xenomai :

```
$ git  clone  https://source.denx.de/Xenomai/xenomai.git
$ cd  xenomai
$ checkout  v3.2.2
```

Appliquer le patch sur le noyau :

```
$ ./scripts/prepare-kernel.sh  --linux=../linux-5.4.228  --ipipe=../ipipe-core-5.4.228-x86-12.patch  --arch=x86_64
```

Configurer le kernel :

```
$ cd ../linux-5.4.228
$ cp /boot/config-$(uname -r) ./.config
$ make menuconfig

Processor type and features  --->
      [ ] Multi-core scheduler support

Power management and ACPI options  --->
      CPU Frequency scaling  --->
            [ ] CPU Frequency scaling
      [*] ACPI (Advanced Configuration and Power Interface) Support  --->
            < >   Processor
      CPU Idle  --->
            [ ] CPU idle PM support

Memory Management options  --->
      [ ] Contiguous Memory Allocator
      [ ] Transparent Hugepage Support
      [ ] Allow for memory compaction
      [ ] Page migration

Device Drivers  --->
      Microsoft Hyper-V guest support  --->
            < > Microsoft Hyper-V client drivers

Cryptographic API  --->
      Certificates for signature checking  --->
            ()    Additional X.509 keys for default system keyring
```

```
Kernel hacking  --->
      Compile-time checks and compiler options  --->
            [ ] Compile the kernel with debug info
            [ ] Tracers  ----
```

Compiler, installer et tester le noyau :

```
$ make  -j 8

$ sudo  make  modules_install

$ sudo  make  install

$ sudo  reboot



$ cat  /proc/xenomai/version
  3.2.2
```

Installer les bibliothèques de Xenomai :

```
$ cd xenomai/

$ ./scripts/bootstrap

$ ./configure --enable-smp --host=i686-linux --enable-pshared  --with-
core=cobalt --enable-x86-vsyscall CFLAGS="-m32 -O2" LDFLAGS="-m32"

$ make

$ sudo make install
```

Test de Xenomai

Consultez les informations fournies par ipipe :

```
$  ls  /proc/ipipe/
Linux  version  Xenomai
```

Consultez les informations fournies par Xenomai :

```
$  ls  /proc/xenomai/
affinity  apc  clock  faults  heap  irq  latency  registry  sched  timer
version
```

Lancez un test de mesure de latence :

```
$  sudo  /usr/xenomai/bin/latency
```

En parallèle, provoquez des perturbations sur le système :

```
$  sudo  /usr/xenomai/bin/dohell  durée_en_secondes

$ top -d 0

$ ping -f 192.168.3.1
```

(depuis une autre machine)

À vous

Essayez `latency` en augmentant progressivement :

- la charge système (nombre de processus actifs)
- la charge en appels-système (par exemple `dd if=/dev/zero of=/dev/null`)
- la charge en interruptions.

Quelles performances pouvez attendre de ce système ?

Programmation sous Xenomai

Les tâches temps-réel de Xenomai peuvent être lancées en contexte utilisateur (le plus simple et le seul traité ici) ou dans un contexte noyau (nécessité d'écrire un module du noyau).

Pour éviter les problèmes de gestion de la mémoire virtuelle, il faut appeler

`mlockall(MCL_CURRENT | MCL_FUTURE)` en début de programme.

Création de tâche :

```
int rt_task_spawn(RT_TASK *task, const char *name,
                  int stack_size, int rt_prio, int attr,
                  void (*function) (void *), void *arg);
```

attributs :

- `T_FPU` : utilisera l'unité de calculs arithmétiques,
- `T_SUSP` : la tâche doit être créée dans un état *Suspendu*
- `T_CPU(`*num*`)` : exécuter la tâche sur un processeur donné
- `T_JOINABLE` : on pourra attendre la fin de la tâche.

Modes primaire et secondaire

Les tâches temps réel ordonnancées par Xenomai s'exécutent en **mode primaire**.

Elles ne peuvent être préemptées que par des tâches Xenomai plus prioritaires ou des handlers d'interruption gérés par Xenomai.

Lorsqu'une tâche temps réel effectue un appel-système elle est ordonnancée par Linux et passe en **mode secondaire**.

Elle peut être préemptée par les tâches et les handlers Linux plus prioritaires.

Pour éviter les commutations en mode secondaire, on se limite aux fonctions de l'API Xenomai. Certains appels système ont été réécrits :

```
#include <rtdk.h>

  int rt_printf(const char *format, ...);
  int rt_fprintf(FILE *fp, const char *format, ...);
  ...
```

Au début du programme, pour initialiser le thread d'affichage, invoquer

`rt_print_auto_init(1);`

```
  int  rt_vfprintf(FILE *fp, const char *format,
                   va_list args);

  int  rt_vprintf(const char *format, va_list args);

  int  rt_puts(const char *s);

  void rt_syslog(int priority, const char *format, ...);

  void rt_vsyslog(int priority, const char *format,
                  va_list args);
```

À vous

Compilez les exemples de ce chapitre.

Examinez le code source de **`exemple-01`**, puis exécutez-le.

Gestion des tâches

Obtenir son identifiant :

```
RT_TASK *rt_task_self(void);
```

Suspendre temporairement l'exécution d'une tâche :

```
int rt_task_suspend(RT_TASK *task);
```

Une tâche suspendue peut être relancée avec :

```
int rt_task_resume(RT_TASK *task);
```

Détruire une tâche :

```
int rt_task_delete(RT_TASK *task);
```

Obtenir l'état d'une tâche :

```
int rt_task_inquire(RT_TASK *task, RT_TASK_INFO *info);
```

Attendre la fin d'une tâche joignable :

```
int rt_task_join(RT_TASK *task);
```

Modifier la priorité :

```
int rt_task_set_priority(RT_TASK *task, int priority);
```

Tâches périodiques

Rendre une tâche périodique :

```
int rt_task_set_periodic(RT_TASK *task, RTIME start,
                         RTIME period);
```

Attendre la période suivante :

```
int rt_task_wait_period(unsigned long *nb_overrun);
```

Céder le processeur à une autre tâche de même priorité :

```
int rt_task_yield(void);
```

Sommeil et attentes

Lire l'heure :

```
RTIME rt_timer_read(void);
```

Attendre :

```
void rt_timer_spin(RTIME delay_ns); // Attente active

int rt_task_sleep(RTIME delay_ns);
int rt_task_sleep_until(RTIME dated);
```

À vous...

Examinez le code source de l'**`exemple-02`**. Il s'agit d'une tâche activée régulièrement, la période étant précisée en microsecondes au démarrage.

Essayez-le programme. Quelle est la fluctuation maximale du timer. Cela correspond-il aux valeurs obtenues avec `cyclictest` ?

Mutex et sémaphores

Création et destruction de mutex :

```
int rt_mutex_create(RT_MUTEX *mutex, const char *name);

int rt_mutex_delete(RT_MUTEX *mutex);
```

Acquisition et libération d'un mutex

```
int rt_mutex_acquire(RT_MUTEX *mutex, RTIME timeout);

int rt_mutex_release(RT_MUTEX *mutex);
```

Création et destruction d'un sémaphore

```
int rt_sem_create(RT_SEM *sem, const char *nom,
                  unsigned long value, int mode);

int rt_sem_delete(RT_SEM *sem);
```

Acquisition et libération d'un sémaphore

```
int rt_sem_p(RT_SEM *sem, RTIME timeout);

int rt_sem_v(RT_SEM *sem);
```

Les sémaphores peuvent avoir différents modes en fonction du dernier argument de `rt_sem_create()`.

À vous...

Observez l'**`exemple-03`**. On mesure le temps de commutation entre deux threads autour d'un mutex. Exécutez le programme. Quelle est la durée d'une commutation ?

L'**`exemple-04`** implémente un scénario d'inversion de priorité. Les mutex de Xenomai disposent automatiquement d'un héritage de priorité pour éviter l'inversion. Vérifiez le comportement du programme.

Messages synchrones

```
ssize_t rt_task_send(RT_TASK * dest,
                     RT_TASK_MCB * mcb_request,
                     RT_TASK_MCB * mcb_reply,
                     RTIME timeout);

int rt_task_receive(RT_TASK_MCB * mcb_r, RTIME timeout);

int rt_task_reply(int exchange_id, RT_TASK_MCB *mcb_reply);
```

Files de messages

```
int rt_queue_create(RT_QUEUE *queue, const char *name,
                    size_t size, size_t maxi, int mode);
int rt_queue_delete(RT_QUEUE *queue);

void * rt_queue_alloc(RT_QUEUE *queue, size_t size);
int    rt_queue_free(RT_QUEUE *queue, void * buffer);

int     rt_queue_send(RT_QUEUE *queue, void * buffer,
                      size_t size, int mode);
ssize_t rt_queue_receive(RT_QUEUE *queue, void **buffer,
                         RTIME timeout);
```