Recherche Bibliographique

Alexandre Benazech, Vincent Pera et Romain Rivière

Abstract

L'objet de ce document est la recherche documentaire menée à propos de Xenomai dans le cadre du projet tutoré de 4IR. Le but premier est de décrire Xenomai 3 et de présenter ses différentes fonctionnalités. Le rapport contient des rappels sur les systèmes d'exploitation temps réel, ainsi que des informations sur la cible utilisée, le Raspberry Pi 3.

Introduction

L'informatique embarquée est présente tout autour de nous à travers les objets qui nous entourent, et cette tendance semble s'accélérer avec l'essor massif de l'Internet des Objets . Cependant, dans certains domaines particuliers tels que celui des transports, les systèmes sont soumis à des contraintes de temps de réponse strictes. En réponse à ces contraintes ont été introduits les systèmes temps réel. Dans le cadre de la réalisation du Segway, nous utiliserons l'OS temps réel Xenomai qui permet aussi d'utiliser les services Linux.

Temps Réel

Définition

D'après Pierre Ficheux dans l'article *Linux embarqué* de *Techniques de l'ingénieur* : *"Un système temps réel est une association logiciel/matériel où le logiciel permet une gestion adéquate des ressources matérielles en vue de remplir certaines tâches dans des limites temporelles bien précises."*.

La complexité entière de ces systèmes repose sur les contraintes de temps qui sont ajoutées à l'exactitude du résultat recherché. Ainsi le résultat doit en effet être exact mais il doit surtout être donné dans un temps imparti bien particulier.

Applications du temps réel

L'application de temps réel s'effectue par l'utilisation d'un Système d'Exploitation en Temps Réel, appelé RTOS (Real-Time Operating System) en anglais, pour les systèmes embarqués tel que les téléphones, les robots, etc… Un OS en temps réel fournit les services et primitives qui peuvent être utilisés pour la création d'un système temps réel, mais il ne garantit pas que le résultat final respecte le temps réel. Cela dépendra uniquement du développement qui se fera dessus.

Il existe une grande variété d'OS temps réel. Nous utiliserons dans le cadre de nos travaux Xenomai 3. Les systèmes temps réel sont utilisés dans de nombreux secteurs bien différents les uns des autres aujourd'hui. Ainsi, les salles de marché d'une bourse sont entièrement des systèmes de temps réel, ce qui parait évident vu la rapidité des variations de la bourse ainsi que des achats – ventes. Un système comme celui-ci doit en effet répondre exactement à la requête envoyée, comme les systèmes normaux, mais surtout dans un temps précis. Une requête

d'achat qui serait validée une heure après l'avoir soumis à notre terminal serait contre-productif.

Les salles de marché d'une bourse

De plus, les systèmes de surveillance d'une centrale nucléaire sont aussi en temps réel. Ainsi les mesures renvoyées par les capteurs doivent arriver dans un temps précis sur les écrans de contrôle, et les actions décidés par les surveillants de la fission doivent être appliquées dans un temps très précis qui ne peut être dépassé. Un dépassement pourrait avoir des conséquences désastreuses.

D'autre part le temps réel est aussi appliqué dans des utilisations qui ne sont pas aussi critiques. Ainsi dans une conversation par visioconférence, le temps réel est très présent pour que les deux participants puissent s'entendre comme s'ils parlaient face à face. Un délai dans ce genre de cas ne sera pas forcément gênant pour les utilisateurs tant que l'application reste utilisable.

Temps réel mou / dur

Nous pouvons donc constater d'après ces trois exemples qu'il existe deux types de temps réel. La différence principale entre ces deux types de temps réel provient de l'importance que l'on apporte aux contraintes temporelles du système que l'on veut intégrer.

Le temps réel dur est comme son nom l'indique, le temps réel le plus strict sur les contraintes temporelles. Il ne tolère aucun dépassement sur ces dernières.

Poste de pilotage d'une navette spatiale

Ce système doit en effet respecter ces contraintes dans le meilleur et surtout dans le pire des cas. Le non-respect de ces contraintes peut amener des situations délicates car ils sont utilisés dans différents secteurs tel que le pilotage d'avion de ligne ou de navettes spatiales, les systèmes nucléaires comme vu auparavant. Les navettes spatiales utilisent des systèmes de temps réel dur pour leurs mesures.

Le temps réel mou ou souple est plus laxiste que le temps réel dur. Il permet de dépasser les contraintes temporelles exceptionnellement dans la limite que le système reste utilisable. On peut retrouver ce genre de temps réel dans les visioconférences tel que Skype ou encore les jeux vidéo en ligne qui autorise un certain décalage temporel sans que l'utilisation devienne gênante.

Temps réel et Linux

Le temps réel sous Linux est initié par la branche Linux-rt pour répondre aux différentes contraintes qu'impose le temps réel. C'est le patch PREEMPT_RT qui intègre le temps réel au noyau Linux et permet de donner à ce dernier le comportement temps réel en limitant le nombre de modifications apportées au noyau . Il rend préemptible la majeure partie du noyau, c'est-à-dire qu'il devient capable d'exécuter ou d'arrêter une tâche planifiée en cours, surtout en ce qui concerne les sections critiques et les interruptions générées par le noyau. De plus, grâce à ce patch, les temps de latence induits par le système sont réduits pour répondre aux contraintes du temps réel dur .

Par rapport à Xenomai, PREEMPT_RT modifie le fonctionnement du noyau alors que Xenomai ajoute un second noyau ou une couche de temps réel, ce qui a pour effet d'alléger le système

résultant. Xenomai passe par la couche de virtualisation Adeos qui est un patch logiciel à installer sur le noyau Linux .

Principe de fonctionnement de Xenomai

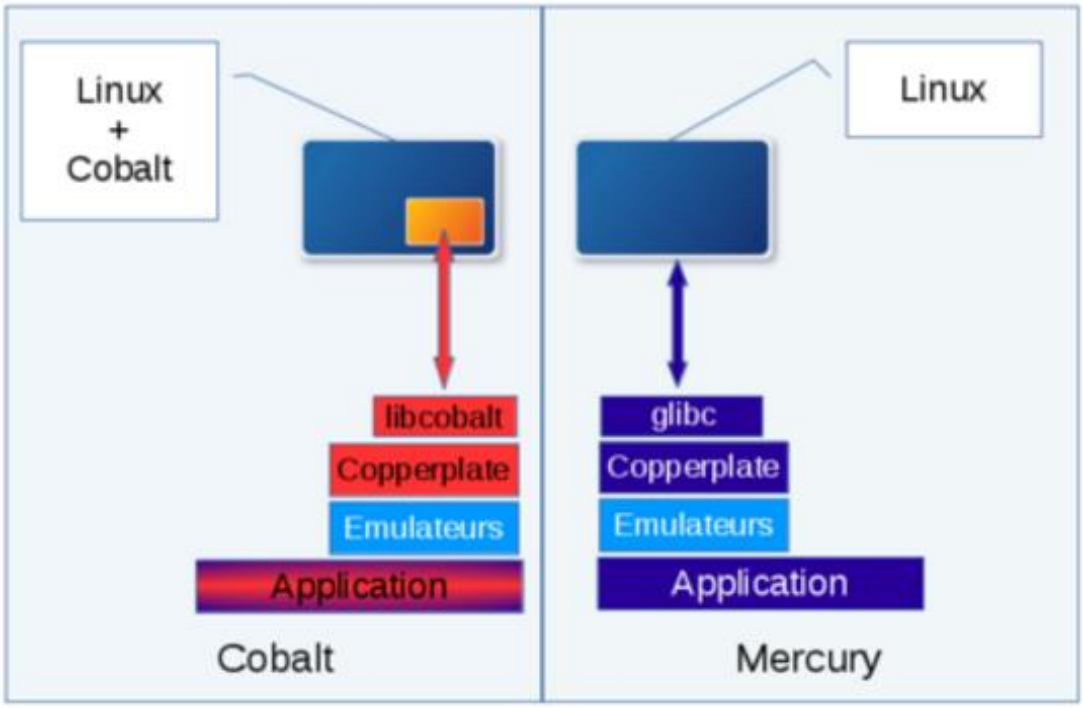
Xenomai est un système d'exploitation temps réel ayant Linux en tâche de fond. Cet OS a pour but d'émuler des RTOS. Ainsi, Linux peut être préempté comme une simple tâche. Xenomai offre donc une garantie d'exécution en temps réel dur pour les tâches qu'il gère. Fondé en 2001 , sa version actuelle est la version 3.0.3. Il existe depuis la version 3 deux variantes de Xenomai: *Mercury* et *Cobalt*.

La nouvelle variante *Mercury*, introduite dans la version 3, se repose sur les services de Linux pour faire fonctionner du code applicatif via les bibliothèques de Xenomai. Il est préférable que la version de Linux dispose de PREEMPT_RT, bien que cela ne soit pas une obligation . En utilisant *Mercury* sur une Linux patché avec PREEMPT_RT on peut considérer le système comme temps réel dur.

La version *Cobalt* reprend l'architecture co-noyau des versions précédentes de Xenomai. Le système est donc constitué de deux noyaux, un noyau Linux et un noyau Cobalt . Cette architecture est largement décrite dans la suite de cette section car c'est l'architecture retenue pour le projet. Elle propose elle aussi un temps réel "dur" mais grâce à son routage d'interruption effectué par le noyau parallèle au noyau Linux on obtient des latences plus faibles qu'avec un routage natif .

Copperplate

Depuis la version 3 de Xenomai, les services d'émulation des RTOS se situent dans l'espace utilisateur, plus précisément dans la bibliothèque de services *Copperplate*. Copperplate permet aussi la médiation entre les API émulées et les services proposés par les noyaux. Dans le cas de l'utilisation de *Cobalt*, *Copperplate* utilisera les services POSIX du co-noyau. Dans le cas de l'utilisation de *Mercury*, Copperplate utilisera glibc et de ses services POSIX natifs.

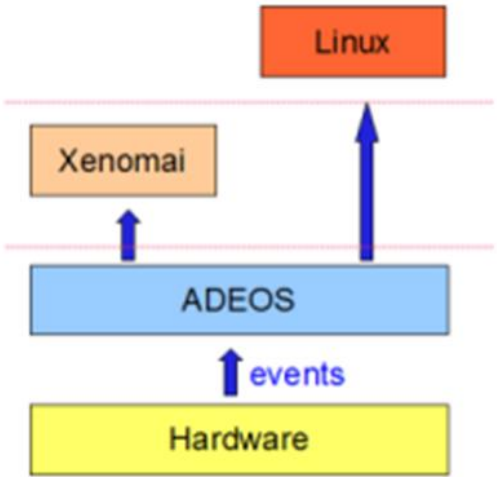


Interface de médiation Copperplate

En pratique l'utilisation de *Copperplate* permet de faire fonctionner des applications sur l'un ou l'autre environnement sans qu'il y ait de changement dans le code .

Le routage des interruptions

Dans le cas de l'utilisation de *Cobalt*, la question posée est de savoir quel noyau va traiter l'interruption entrante.



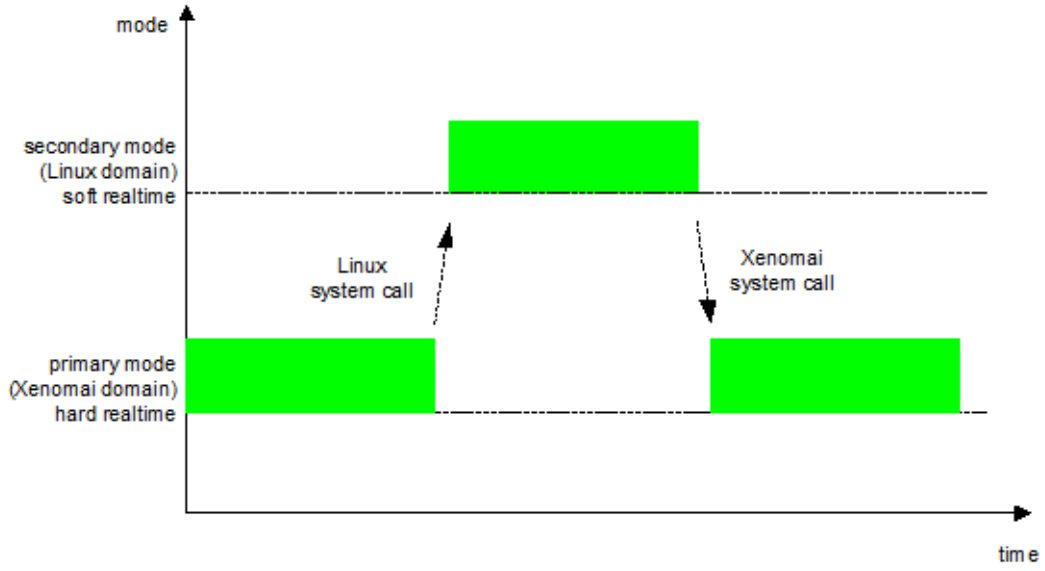
La gestion évènementielle avec ADEOS

Pour maîtriser le traitement des évènements, une couche appelée ADEOS est intercalée entre le matériel et les deux systèmes d'exploitation.

ADEOS est un nanokernel qui va capturer les évènements pour les diriger, dans un second temps, vers Xenomai ou Linux. Cependant, un ordre est respecté pour répondre toujours aux contraintes du temps réel. Xenomai traite l'évènement entrant en premier. Dans le cas où celui-ci ne lui serait pas destiné, l'évènement serait dirigé vers Linux.

Le partage des ressources

Lorsque nous utilisons *Cobalt*, nous avons deux OS, chacun possédant son propre scheduler. Les tâches Xenomai sont donc dans le scheduler de Xenomai, les tâches Linux dans le scheduler Linux. Cependant, les tâches Xenomai peuvent être amenées à accéder à une ressource appartenant au noyau de Linux. Il est impossible d'aller la chercher directement depuis Xenomai car seul Linux peut vérifier sa validité.
Il faut donc passer par Linux via un appel système Linux. Pour traiter ces appels, les tâches sont déplacées d'un scheduler à un autre en fonction des ressources accédées. Le mécanisme dit de Shadow threading permet l'exécution des tâches Xenomai sous Linux.



Migration d'une tâche

Une tâche est dite en primary mode si elle est schedulée par Xenomai. Une tâche Xenomai schedulée par Linux est dite en secondary mode.

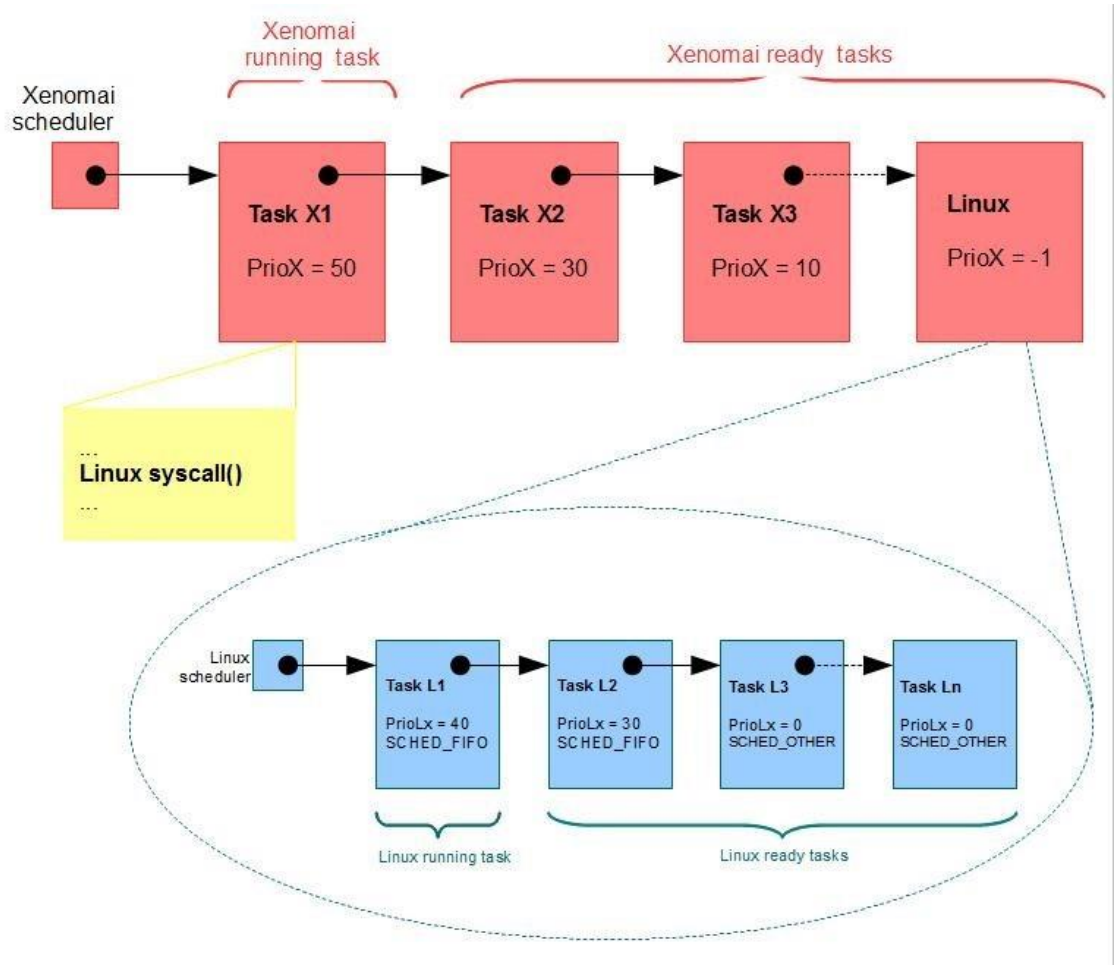
Une tâche Xenomai commence en primary mode. Puis, à la suite d'un appel système Linux, elle bascule en secondary mode. Elle rebascule en primary mode après une appel système Xenomai.

La gestion des priorités en primary et secondary mode

Il faut affecter une certaine priorité aux tâches Xenomai passées dans le scheduler de Linux et à la tâche Linux elle-même (qui se trouve dans le scheduler de Xenomai).

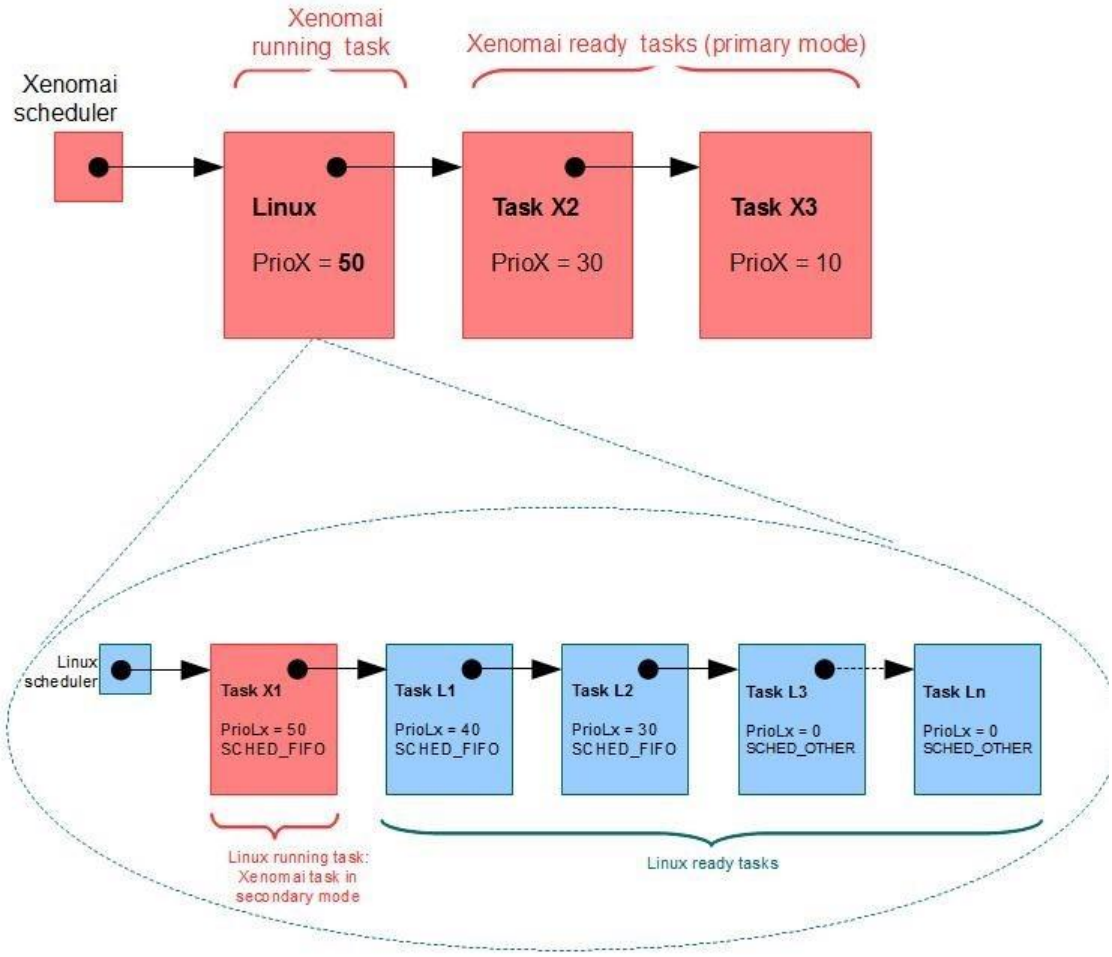
L'implémentation de Xenomai est telle qu'une tâche Xenomai est à priorité constante, quel que soit le scheduler dans lequel elle se trouve. Pour éviter les cas d'inversion de priorité, il faut que

la priorité de la tâche Linux dans le scheduler Xenomai évolue dynamiquement. Elle est égale à la priorité maximale des tâches en secondary mode, ou à -1 s'il n'y a aucune tâche en secondary mode. Il faut également préciser que les tâches passées en secondary mode ne répondent que à des contraintes temps réel mou. Pour les tâches à contraintes temps réel dur, il faut s'assurer qu'elles restent en primary mode.
A travers les différents schémas qui vont suivre, nous allons présenter l'évolution des priorités telle qu'elle est faite avec Xenomai. Nous notons Ln les tâches schedulées par Linux avec une priorité notée PrioLx, Xn les tâches schedulées par Xenomai avec une priorité notée PrioX.



Linux et tâche de fond

La tâche X1 de priorité 50 fait un appel système Linux. La tâche Linux du scheduler de Xenomai est avec une priorité -1 (car aucune tâche n'est en secondary mode). On remarque par ailleurs que deux tâches sont en état « run » en même temps, le choix se faisant en fonction du scheduler et non du processeur (donc une tâche choisie par scheduler).



Evolution dynamique de la priorité de la tâche Linux

La tâche X1 passe dans le scheduler de Linux en gardant sa priorité (50). La tâche Linux du scheduler de Xenomai passe avec une priorité de 50. Ainsi, on supprime le risque d'inversement de priorité, la tâche X1 sera bien faite avant les tâches Xenomai de priorité plus faible.

Il est à noter que si une tâche du scheduler Linux avait une plus grande priorité que 50, elle préempterait tout le monde.

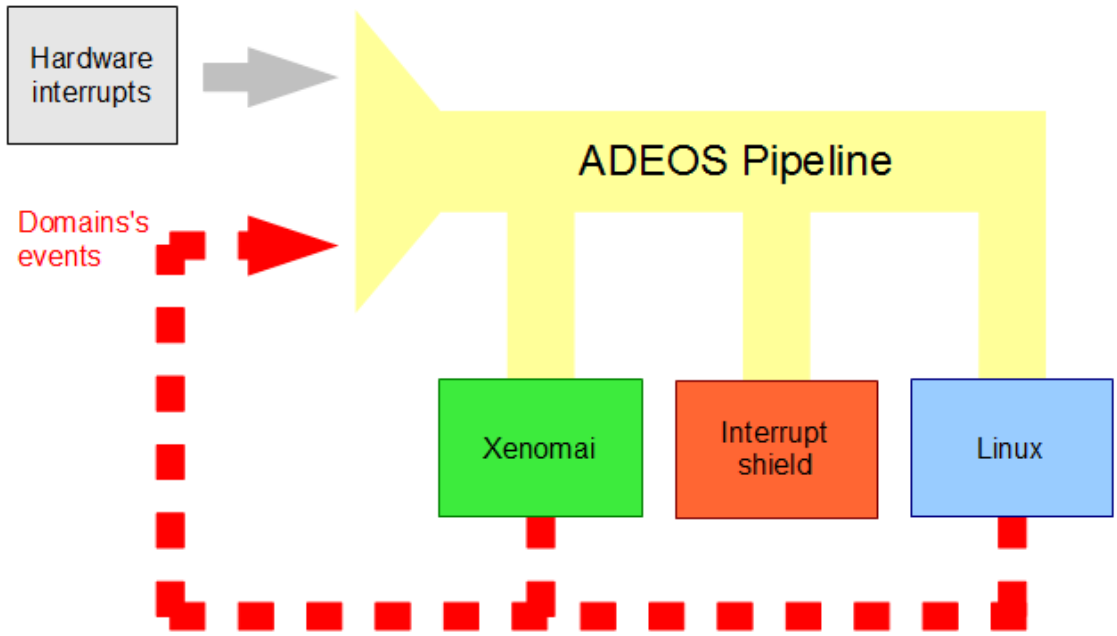
Le routage des évènements avec ADEOS

Nous avons déjà abordé le routage des interruptions mais une interruption peut être généralisée à quelque chose de plus large.

Les évènements peuvent être de plusieurs sortes :

- Interruption matérielle (timer ou périphérique prêt par exemple)
- Signaux logiciels
- Création d'une tâche ou destruction d'une tâche existante

- Changement dans les caractéristiques d'une tâche (changement de priorité par exemple)

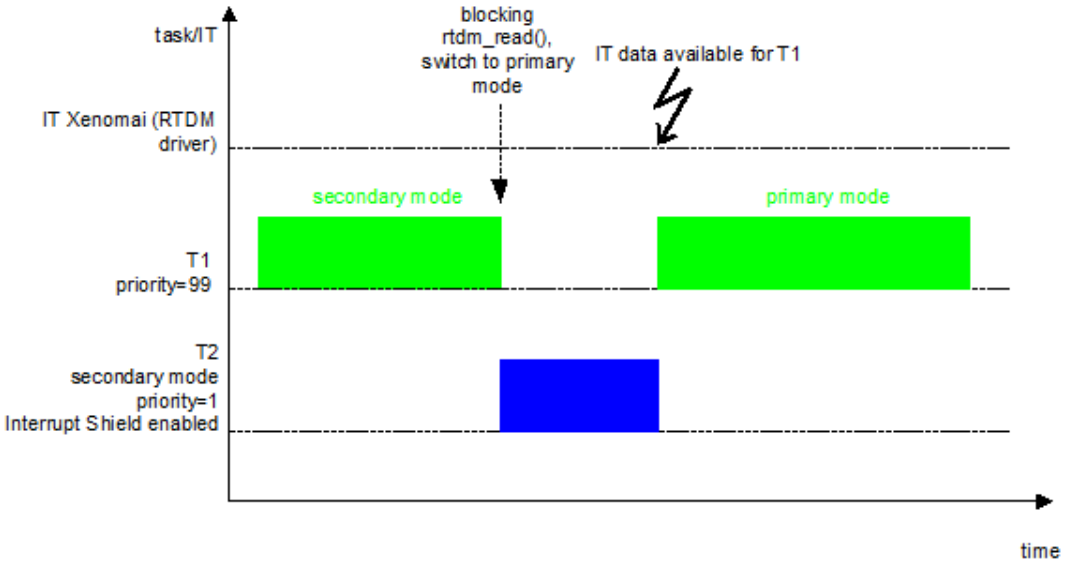


Gestion des évènements sous ADEOS

Le traitement d'un évènement par ADEOS est uniforme peu importe sa nature. L'évènement est capturé par ADEOS et distribué soit à Xenomai, soit à Linux soit à l'Interrupt Shield.

L'Interrupt Shield

Le mécanisme Interrupt Shield protège les tâches en secondary mode en filtrant les interruptions. Leur temps d'exécution reste donc borné.



Gestion des interruptions

Dans l'exemple ci-dessus, nous avons deux tâches T1 et T2 en secondary mode. T1 ayant une plus grande priorité, c'est elle qui a la main en premier. Lorsque T1 se place en attente de donnée, elle est déschedulée et T2 prend la main. Une interruption arrive pour signaler que T1 peut reprendre. L'interruption n'est pas traitée par Linux mais directement par Xenomai. T2 est seulement protégée des IT Linux, mais comme c'est une IT Xenomai, T1 reprend la main et va se terminer avant T2. L'ordre des priorités est donc respecté.

Si l'IT n'avait pas été une IT Xenomai mais une IT Linux, T2 aurait été protégée, l'IT aurait été prise en compte à la terminaison de T2 et donc T2 se serait terminée avant T1. Il y'aurait eu inversion de priorité.

Services offerts par Xenomai

Gestion des tâches et ordonnancement

L'objet central de Xenomai sont les tâches. Xenomai nous permet donc de gérer ces tâches (création, suppression, changement d'état et de propriété).

Xenomai possède un scheduler qui détermine quelle tâche doit s'exécuter à un moment donné. Le scheduler de Xenomai est basé sur un système de priorité, où à chaque tâche on associe une priorité (qui peut être amenée à changer au cours de l'exécution). Xenomai intègre un système avec 99 niveaux de priorité.

Le scheduler de Xenomai utilise un ordonnancement préemptif des tâches. Les politiques d'ordonnancement supportées par Xenomai sont des alternatives de FIFO et Round-Robin basées sur le système de priorité.

La communication inter-tâche

Xenomai étant un système multitâche, les tâches sont en concurrence mais doivent cependant communiquer entre elles pour partager ou échanger des informations et pour se synchroniser. Tout ceci est essentiel pour assurer le bon fonctionnement global de l'application.

Communication par mémoire partagée

Dans cette configuration, une même ressource peut être accédée par plusieurs tâches. Mais il faut garantir un accès exclusif pour les tâches (ie une seule tâche accède à la fois). Pour ce faire, des sections critiques sont utilisées. Xenomai incorpore des mécanismes d'exclusion mutuelle (mutex) permettant la protection de ces sections critiques.

A l'arrivée dans une section critique, le processus se place en attente du mutex (si le mutex est déjà pris) jusqu'à ce que le mutex soit disponible pour ce processus. Le mutex est alors pris par le processus et libéré à la fin de la section critique. Un mutex est une sorte de sémaphore binaire amélioré. Comme un sémaphore binaire, il peut être pris par une unique tâche mais il va en plus gérer seul les changements de ses propriétés et les timeouts.

Communication par file de messages

Xenomai intègre un service de file de messages permettant la communication entre les processus. Une file de message est créée par une tâche et peut être utilisée par plusieurs tâches pour l'échange ou la transmission de données et d'informations.

Cette file est de taille variable et peut contenir des messages de différentes formes.

Synchronisation entre les tâches

Dans un système multitâche tel que Xenomai, les tâches temps-réel nécessitent, dans beaucoup de cas, de se synchroniser avec la terminaison d'un traitement, pour l'accès à une ressource ou encore pour garantir un ordre précis.

Xenomai fournit différents moyens permettant de faire ces synchronisations. Nous avons vu dans le paragraphe précédent un premier moyen avec l'utilisation d'un mutex. Mais d'autres moyens sont disponibles.

Les drapeaux évènement

Un event flag permet de signaler aux tâches l'arrivée d'un évènement particulier, de nature quelconque. Un drapeau est associé à un évènement unique. Il faudra donc plusieurs flags si l'on souhaite surveiller plusieurs évènements.

Le flag peut avoir deux états : soit le drapeau est levé (bit à 1) donc l'évènement s'est bien produit, soit il est baissé (bit à 0) donc l'évènement ne s'est pas encore produit.

Les variables condition

Une variable condition est une variable qui va suspendre une tâche tant qu'une certaine condition n'est pas vérifiée. Lorsque cette condition devient vraie, la tâche redevient exécutable.

Les sémaphores

Un sémaphore va être capable, suite à l'arrivée d'un évènement, de réveiller une tâche particulière ou toutes les tâches qui sont en attente. Contrairement au mutex vu précédemment, le sémaphore peut être détenu par plusieurs tâches en même temps.

Concrètement, un sémaphore à une valeur initiale. Lorsque qu'une tâche souhaite prendre le sémaphore, deux cas se présentent. Si le sémaphore est positif, la tâche rentre dans sa section critique et le sémaphore est décrémenté de 1. A la fin de la section critique, le sémaphore est incrémenté de 1.

Si le sémaphore est à 0, la tâche est placée dans la file d'attente du sémaphore avec des politiques d'ordonnancement tels que FIFO ou basées sur les priorités.

Les timers

Xenomai intègre des mécanismes permettant de mesurer des temps à l'aide des tops d'horloge.

On peut faire une utilisation directe des timers pour rendre des tâches périodiques.

Au-delà de cette simple application, Xenomai met à disposition des alarmes, qui sont des timers généraux.

Toutes les tâches Xenomai ont la possibilité de déclencher des alarmes et de les combiner avec des handlers définis par l'utilisateur. Ce handler est exécuté une fois que le délai spécifié dans l'alarme est écoulé. Les alarmes peuvent être périodiques, auquel cas le réarmement est automatique.

Les watchdogs sont un cas particulier des alarmes car ils se terminent par le réveil d'une tâche en attente au lieu de l'exécution d'un handler.

Utilisation de Xenomai

Cette partie est une présentation des principales primitives contenues dans le module Alchemy API fourni dans Xenomai 3 permettant de mettre en œuvre les services présentés dans la partie précédente. Toutes les fonctions présentées retournent un entier. Cet entier vaut 0 s'il n'y a eu aucune erreur durant le déroulement de la fonction, sinon il correspondra à un code d'erreur. Dans la pratique, il faut tester ce code de retour à chaque fonction pour veiller au bon fonctionnement global. Ces primitives sont issues de la documentation de l'API de Xenomai disponible sur le site officiel de Xenomai .

Management d'une tâche

Une tâche est constituée d'un descripteur, d'une pile d'exécution, d'une priorité et d'un mode. En pratique, le descripteur est de type `RT_TASK` est doit être déclaré avant la création de la tâche.

- La tâche doit être créée avec `rt_task_create` puis lancée avec `rt_task_start`. On peut également utiliser `rt_task_spawn` qui crée et lance la tâche.
- Pour supprimer une tâche, on utilise la fonction `rt_task_delete`. Pour détruire la tâche dès sa terminaison, on peut aussi la lancer en mode `T_JOINABLE` et utiliser la fonction `rt_task_join`.
- Pour modifier la priorité d'une tâche, il suffit d'utiliser la fonction `rt_task_set_priority`.
- Pour suspendre une tâche et reprendre une tâche suspendue, il faut utiliser respectivement `rt_task_suspend` et `rt_task_resume`.
- Pour mettre en attente une tâche, on peut soit utiliser `rt_task_sleep` qui va mettre la tâche en attente jusqu'à ce que le délai passé en paramètre se soit écoulé. On peut aussi utiliser `rt_task_sleep_until` qui va mettre la tâche en attente jusqu'à ce que la date passée en paramètre soit atteinte.

- Pour réveiller une tâche en attente, il faut utiliser `rt_task_unblock`.

Toutes les fonctions présentées ci-dessous sont disponibles dans le fichier task.h.

- **`int rt_task_create (RT_TASK * task, const char * name, int stksize, int prio, int mode) ;`** => Créer une tâche.
- **`int rt_task_start (RT_TASK * task, void (*) (void * cookie) entry, void * cookie) ;`** => Démarrer une tâche
- **`int rt_task_suspend (RT_TASK * task) ;`** => Suspendre une tâche
- **`int rt_task_resume (RT_TASK * task) ;`** => Reprendre une tâche
- **`int rt_task_delete (RT_TASK * task) ;`** => Supprimer une tâche
- **`int rt_task_set_priority (RT_TASK * task, int prio) ;`** => Changer la priorité de la tâche
- **`int rt_task_sleep (RTIME delay) ;`** => Passer une tâche en sommeil (temps relatif)
- **`int rt_task_sleep_until (RTIME date) ;`** => Passer une tâche en sommeil (temps absolu)
- **`int rt_task_unblock (RT_TASK * task) ;`** => Réveiller une tâche
- **`int rt_task_inquire (RT_TASK * task, RT_TASK_INFO * info) ;`** => Obtenir les informations concernant la tâche
- **`int rt_task_spawn (RT_TASK * task, const char * name, int stksize, int prio, int mode, void (*) (void * cookie) entry, void * cookie) ;`** => Créer et lancer une tâche
- **`int rt_task_join (RT_TASK * task) ;`** => Attendre la terminaison de la tâche

Listes des paramètres :

- **`task`** : adresse du descripteur Xenomai de la tâche utilisé pour stocker les données liées à la tâche (identifiant de la tâche)
- **`name`** : nom de la tâche
- **`stksize`** : taille (en octet) de la pile. (0 pour utiliser la taille prédéfinie)
- **`prio`** : priorité de la tâche (entre 0 et 99 avec 0 pour priorité la plus basse)
- **`mode`** : mode de création de la tâche. Les principales valeurs sont :

  - T_FPU : autorise la tâche à utiliser l'unité de calcul à virgule flottante si elle est présente sur la machine
  - T_SUSP : la tâche démarre en mode suspendu
  - T_CPU(cpuid) : déclare une affinité entre la tâche et le processeur numéro cpuid.
  - T_JOINABLE : autorise une autre tâche à attendre la terminaison de la nouvelle tâche
  - 0 = aucun mode particulier
  - Possibilité de concaténer les modes (exemple : T_FPU | T_JOINABLE)

- **entry** : adresse de la fonction C qui doit être exécutée lorsque la tâche temps réel est exécutée.
- **delay** : nombre de tops d'horloge avant le reprise de la tâche
- **date** : date absolue (en tops d'horloge) à partir de laquelle la tâche va reprendre. TM_INFINITE est aussi une valeur permise
- **info** : adresse d'une structure contenant les informations sur la tâche

Communication par mémoire partagées et protection via exclusion mutuelle

En pratique, la communication par mémoire partagée va se faire en déclarant des structures de données en variable globale.

Pour assurer des accès exclusifs aux ressources pour les tâches, on utilise un mutex. Il faut que ce mutex soit visible des différentes tâches qui voudront accéder à la ressource.

Lorsque l'on atteint une section critique, il faut utiliser `rt_mutex_acquire` pour attendre sur le mutex jusqu'à l'obtenir. A la fin de la section critique, il faut utiliser `rt_mutex_release` pour libérer le mutex.
Toutes les fonctions présentées ci-dessous sont disponibles dans le fichier mutex.h.

- **`int rt_mutex_create (RT_MUTEX * mutex, const char * name) ;`** => Créer le mutex
- **`int rt_mutex_delete (RT_MUTEX * mutex) ;`** => Supprimer le mutex
- **`int rt_mutex_acquire (RT_MUTEX * mutex, RTIME timeout) ;`** => Prendre le mutex. Si le mutex est indisponible, on l'attend jusqu'à expiration du timeout (TM_INFINITE est une valeur permise).
- **`int rt_mutex_release (RT_MUTEX * mutex) ;`** => Libérer le mutex

- **`int rt_mutex_inquire (RT_MUTEX * mutex, RT_MUTEX_INFO * info) ;`** => Obtenir le statut du mutex

Liste des paramètres :

- **`mutex`** : adresse du descripteur Xenomai du mutex
- **`name`** : nom du mutex
- **`timeout`** : temps d'attente (en tops d'horloge)
- **`info`** : adresse d'une structure contenant les informations sur le mutex

Communication par file de messages

Toutes les fonctions présentées ci-dessous sont disponibles dans le fichier queue.h.

- **`int rt_queue_create (RT_QUEUE * q, const char * name, size_t poolsize, size_t qlimit, int mode) ;`** => Créer une file de message
- **`int rt_queue_delete (RT_QUEUE * q) ;`** => Supprimer une file de message.
- **`void * rt_queue_alloc (RT_QUEUE * q, size_t size) ;`** => Allouer la mémoire pour accueillir un message dans la file
- **`int rt_queue_free (RT_QUEUE * q, void * buf) ;`** => Libèrer la mémoire d'un tampon dans la file
- **`int rt_queue_send (RT_QUEUE * q, void * mbuf, size_t size, int mode) ;`** => Envoyer un message dans une file
- **`ssize_t rt_queue_receive (RT_QUEUE * q, void ** bufp, RTIME timeout) ;`** => Recevoir un message depuis le file
- **`int rt_queue_flush (RT_QUEUE * q) ;`** => Supprimer tous les messages non lus de la file
- **`int rt_queue_inquire (RT_QUEUE * q, RT_QUEUE_INFO * info) ;`** => Obtenir les informations concernant la file

Liste des paramètres :

- **`q`** : adresse du descripteur Xenomai de la file de message
- **`name`** : nom de la file de message
- **`poolsize`** : taille maximale (en octets) pour chaque message

- **qlimit** : nombre maximal de messages dans la file
- **mode** : mode de création des messages dans la file
- **size** : la taille (en octet) de la mémoire à allouer pour un message
- **buf** : adresse du message à libérer
- **mbuf** : adresse de la mémoire tampon où envoyer le message
- **bufp** : pointeur sur l'emplacement de mémoire prévu pour la réception du message
- **timeout** : délai en tops d'horloge d'attente de réception d'un message (`TM_INFINITE` est une valeur permise)
- **info** : adresse d'une structure contenant les informations sur la file de message

Synchronisation par drapeaux évènementiels

Toutes les fonctions présentées ci-dessous sont disponibles dans le fichier event.h.

- **`int rt_event_create (RT_EVENT * event, const char * name, unsigned long ivalue, int mode) ;`** => Créer un groupe de flag
- **`int rt_event_delete (RT_EVENT * event) ;`** => Supprimer un groupe de flag
- **`int rt_event_signal (RT_EVENT * event, unsigned long mask) ;`** => Mettre à jour le groupe de flag
- **`int rt_event_wait (RT_EVENT * event, unsigned long mask, unsigned long * mask_r, int mode, RTIME timeout) ;`** => Attendre que un ou plusieurs flags soient levés
- **`int rt_event_clear (RT_EVENT * event, unsigned long mask, unsigned long * mask_r) ;`** => Vider un groupe de flag
- **`int rt_event_inquire (RT_EVENT * event, RT_EVENT_INFO * info) ;`** => Obtenir des informations concernant un groupe de flags

Liste des paramètres :

- **event :** adresse du descripteur Xenomai d'une groupe de flag.
- **name :** nom du groupe de flag
- **ivalue :** valeur initiale du masque pour le groupe de flag

- **mode :** mode de création (dans `rt_event_create`) mode d'attente : EV_ANY pour une attente sur un flag levé dans mask, `EV_ALL` pour une attente sur tous les flags levés dans mask (dans `rt_event_wait`)
- **mask :** valeur pour la mise à jour (dans `rt_event_signal`) valeur attendue pour le groupe de flag (dans `rt_event_wait`) flags qui doivent être baissés (dans `rt_event_clear`)
- **mask_r :** adresse pour sauvegarder la valeur du masque au moment où la tâche passe en état exécutable (dans rt_event_wait) adresse pour sauvegarder la valeur des flags avant le clear (dans rt_event_clear)
- **timeout :** délai (en tops d'horloge) d'attente (`TM_INFINITEFT` est une valeur permise)
- **info :** adresse d'une structure contenant les informations sur le groupe de flags

Synchronisation par variables condition

La fonction `rt_cond_wait` est bloquante et permet de lancer une attente sur la variable condition. La tâche qui doit être exécutée (d'après la politique d'ordonnancement de la file d'attente associée à la variable condition) reprend lorsqu'une autre tâche exécute `rt_cond_signal.`

Si c'est `rt_cond_broadcast` qui est exécutée, toutes les tâches de la file d'attente associée passent dans l'état exécutable.

Toutes les fonctions présentées ci-dessous sont disponibles dans le fichier cond.h.

- **`int rt_cond_create (RT_COND * cond, const char * name) ;`** => Créer une variable condition
- **`int rt_cond_delete (RT_COND * cond) ;`** => Supprimer une variable condition
- **`int rt_cond_signal (RT_COND * cond) ;`** => Débloquer la première tâche en attente dans la file associée à la variable condition
- **`int rt_cond_broadcast (RT_COND * cond) ;`** => Débloquer toutes les tâches en attentes dans la file associée à la variable condition
- **`int rt_cond_wait (RT_COND * cond, RT_MUTEX * mutex, RTIME timeout) ;`** => Attendre sur la variable condition
- **`int rt_cond_inquire (RT_COND * cond, RT_COND_INFO * info) ;`** => Obtenir des informations concernant la variable condition

Liste des paramètres :

- **cond :** adresse du descripteur Xenomai de la variable condition

- **name :** nom de la variable condition
- **mutex :** adresse du descripteur Xenomai du mutex qui protège la variable condition
- **timeout :** délai (en tops d'horloge) d'attente (`TM_INFINITE` est une valeur permise)
- **info :** adresse d'une structure contenant des informations sur la variable condition

Synchronisation par sémaphores

Les utilisations des sémaphores se déroulent en trois étapes :

- Créer le sémaphore avec `rt_sem_create`
- Attendre le sémaphore avec `rt_sem_p` jusqu'à avoir le sémaphore
- Libérer le sémaphore après son utilisation avec `rt_sem_v`

Toutes les fonctions présentées ci-dessous sont disponibles dans le fichier sem.h.

- **`Int rt_sem_create (RT_SEM * sem, const char * name, unsigned long icount, int mode) ;`** => Créer le sémaphore
- **`int rt_sem_delete (RT_SEM * sem) ;`** => Supprimer un sémaphore
- **`int rt_sem_p (RT_SEM * sem, RTIME timeout) ;`** => Prendre le sémaphore
- **`int rt_sem_v (RT_SEM * sem) ;`** => Libérer le sémaphore
- **`int rt_sem_inquire (RT_SEM * sem, RT_SEM_INFO * info) ;`** => Obtenir des informations concernant le sémaphore

Liste des paramètres :

- **sem :** adresse du descripteur Xenomai du sémaphore
- **name :** nom du sémaphore
- **icount :** valeur initiale du compteur
- **mode :** politique d'ordonnancement de la file d'attente associée au sémaphore. `S_FIFO` pour politique FIFO, `S_PRIO` pour politique sur un système de priorités
- **timeout :** délai (en tops d'horloge) d'attente du sémaphore (`TM_INFINITE` est une valeur permise)
- **info :** adresse d'une structure contenant des informations sur le sémaphore

Rendre une tâche périodique

Pour la création d'une tâche périodique, il faut utiliser les mêmes fonctions que présentées précédemment. En revanche, il faut utiliser la fonction `rt_task_set_periodic` pour préciser la périodicité de la tâche. Il faudra aussi utiliser `rt_task_wait_period` pour déscheduler la tâche durant l'attente pour la prochaine période. Les autres fonctions applicables sont identiques à celle présentées précédemment.
Toutes les fonctions présentées ci-dessous sont disponibles dans les fichiers task.h.

- **`int rt_task_set_periodic (RT_TASK * task, RTIME idate, RTIME period) ;`** => Indique qu'une tâche est périodique ainsi que sa période
- **`int rt_task_wait_period (unsigned long * overruns_r) ;`** => Attendre jusqu'à la prochaine période

Liste des paramètres :

- **`task :`** adresse du descripteur Xenomai de la tâche utilisé pour stocker les données liées à la tâche (identifiant de la tâche)
- **`idate :`** date initiale (date absolue) en tops d'horloge. `TM_NOW` donne le temps système courant
- **`period :`** période de la tâche en tops d'horloge.

Réglages des Timers

`Rt_timer_inquire` permet d'acquérir des informations sur le timer.

Toutes les fonctions présentées ci-dessous sont disponibles dans le fichier timer.h

- **`int rt_timer_inquire (RT_TIMER_INFO * info) ;`** => Obtenir des informations concernant le timer
- **`RTIME rt_timer_read(void) ;`** => Lire le temps courant

Liste des paramètres :

- **`info :`** adresse d'une structure contenant les informations du timer

Les alarmes

Toutes les fonctions présentées ci-dessous sont disponibles dans le fichier alarm.h.

- **`int rt_alarm_create (RT_ALARM * alarm, const char * name) ;`** => Créer une alarme
- **`int rt_alarm_delete(RT_ALARM * alarm) ;`** => Supprimer une alarme

- **`int rt_alarm_start (RT_ALARM * alarm, RTIME value, RTIME interval) ;`** => Activer une alarme
- **`int rt_alarm_stop (RT_ALARM * alarm) ;`** => Désactiver l'alarme
- **`int rt_alarm_inquire (RT_ALARM * alarm, RT_ALARM_INFO * info) ;`** => Obtenir des informations sur l'alarme
- **`int rt_alarm_wait (RT_ALARM * alarm) ;`** => Attendre la prochaine alarme

Liste des paramètres :

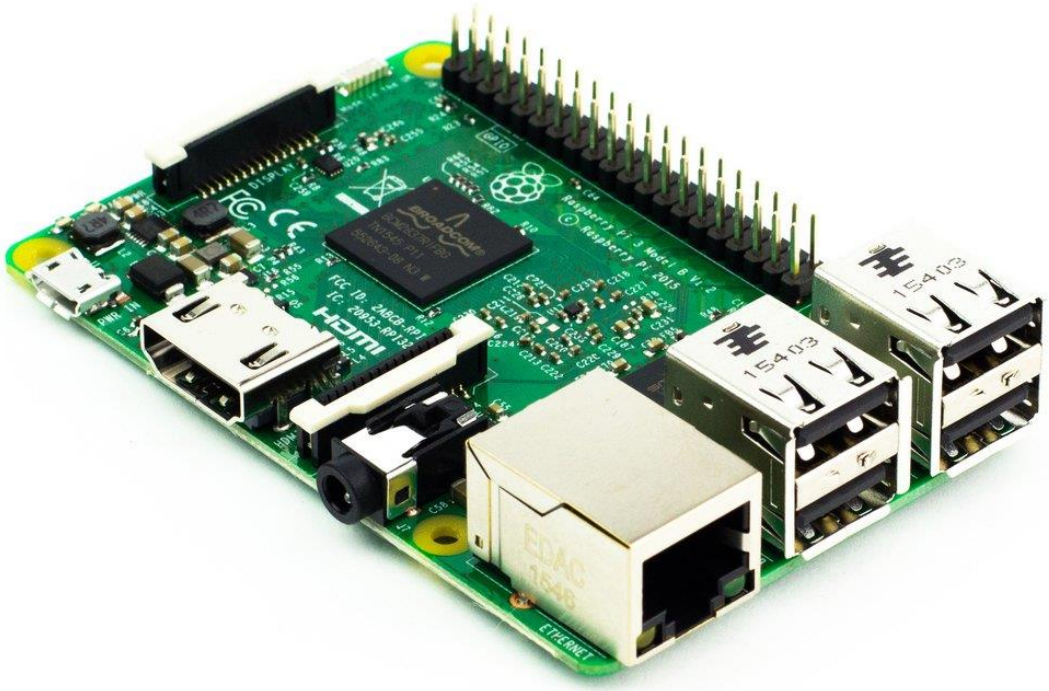
- **`alarm :`** adresse du descipteur Xenomai d'une alarme
- **`name :`** nom de l'alarme
- **`value :`** date relative (en tops d'horloge) du premier déclanchement
- **`interval :`** durée (en tops d'horloge) du réarmement de l'alarme. Mettre `TM_INFINITE`pour les alarmes ONE-SHOT.
- **`info :`** adresse d'une structure contenant les informations sur l'alarme

Raspberry Pi

Qu'est-ce qu'un Raspberry Pi ?

Le Raspberry Pi est un nano-ordinateur monocarte dont le premier modèle est sorti en 2012. Bien qu'il soit de dimensions très réduites, il propose des fonction nalités équivalentes à un ordinateur « classique ». Initialement créé pour promouvoir l'enseignement de l'informatique dans les pays en développement, le Raspberry pi a rencontré un grand succès grâce à son faible coût et à sa grande polyvalence.

Il existe désormais plusieurs itérations de ce produit, les dernières en date étant le Raspberry Pi 3 Model B qui sera utilisé dans le projet ainsi que le Rasperry zero.



La carte Raspberry Pi 3

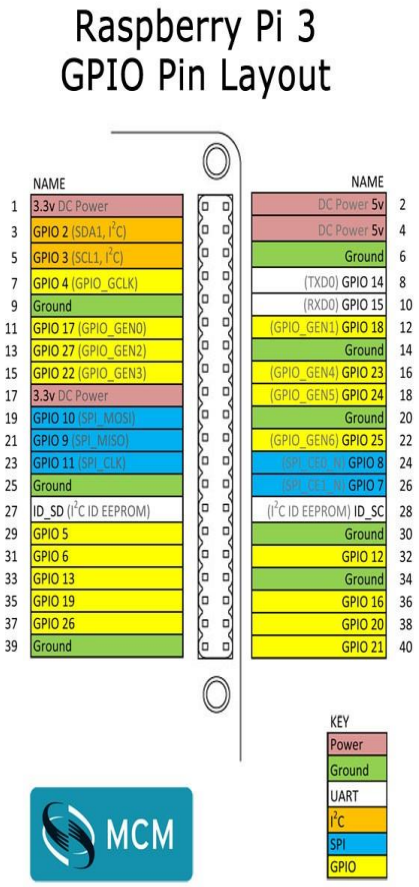
Le Raspberry Pi Model 3

Le Raspberry Pi model 3 permet d'exécuter des systèmes d'exploitation de type GNU/Linux permettant différents types de temps réels allant du temps réel réel mou avec un noyau Linux Vanilla jusqu'au temps réel dur tels qu'un noyau linux patché avec PREEMPT_RT . Il est aussi possible d'utiliser Xenomai.
Le Raspberry Pi 3 dispose d'un processeur Broadcom BCM2837 64 bit à quatre cœurs ARM Cortex-A53 à 1,2 GHz et de 1 Go de mémoire vive. Il dispose aussi d'une connectique assez complète pour sa taille, grâce à 4 ports USB 2.0, un port ethernet, d'un port HDMI, d'un port jack 3.5mm et depuis la dernière version il dispose aussi du Wifi intégré ainsi que du Bluetooth 4.1 LE .

Le Raspberry Pi 3 dispose aussi d'un port GPIO de 40 pins permettant de lire et d'écrire des signaux digitaux. Pour générer un signal analogique,il est nécessaire d'utiliser des signaux PWM.

Il faut aussi avoir recours à un Convertisseur Analogique/Numérique pour lire des signaux analogiques. Cela ne pose pas de problème dans le cadre du projet puisqu'aucun signal analogique n'aura à être lu.

Le GPIO dispose aussi de ports spéciaux tels que les ports GPIO9, GPIO10 et GPIO11 pouvant être utilisés dans le cadre d'une communication via un bus SPI qui sera utilisé dans notre projet. Il dispose aussi entre autres de pins gérant l'I2C. Chaque GPIO configuré comme entrée peut être utilisé comme source d'interruption pour l'ARM. Le GPIO est présenté dans la figure suivante.



Le GPIO du Raspberry Pi 2

Le Raspberry est alimenté via un connecteur micro USB délivrant 5V. Il est recommandé d'utiliser une alimentation pouvant supporter un courant de 2,5A. En IDLE le raspberry consomme environs 300mA et peut monter à 1,3A en stress-test .

Raspberry et Xenomai

Il est possible d'installer Xenomai 3 sur un Raspberry Pi 3. Il n'existe pas encore de version 64 bits mais il est parfaitement possible d'installer la version 32 bits en ayant quand même le gain

de performance procuré par le Raspberry Pi 3 par rapport au 2. La procédure d'installation est décrite par Christophe Blaess sur son blog .

Conclusion

Durant ce travail de recherche bibliographique nous avons pu en apprendre davantage sur les OS temps réel et plus précisément sur la version 3 de Xenomai. Cet OS, notamment grâce à son architecture co-noyau permet de disposer d'un OS Linux qui réponde aux contraintes de temps réel "dur". De plus dans ce document nous avons pu établir un inventaire des services de Xenomai ainsi qu'une description de l'API. Nous pourrons donc nous y référer facilement lors de la réalisation de notre projet.

ETIEMBLE, Daniel, Introduction aux systèmes embarqués, enfouis et mobiles **In:** *Techniques de l'ingénieur*, (Ref: H8000 V2), 2016.

FICHEUX, Pierre, Linux Embarqué **In:** *Techniques de l'ingénieur*, (Ref: H1570 V1), 2010.

CLARK, Libby, Intro to Real-Time Linux for Embedded Developers **In:** *Linux.com, News for the Open Source Professional* **[en ligne].** Mis en ligne le 21/03/2013. Disponible sur : https://www.linux.com/blog/intro-real-time-linux-embedded-developers (Consulté le 25/01/2017)

Systems based on Real time preempt Linux **In:** *Systems based on Real time preempt Linux*, https://rt.wiki.kernel.org/index.php/Systems_based_on_Real_time_preempt_Linux(Consulté le 30/01/2017)

PAGETTI, Claire, Introduction au Temps réel, **In:** *Systèmes temps réel: langages de programmation de systèmes temps réel* , http://www.onera.fr/sites/default/files/u490/cours_temps_reel_in2-nup-nup.pdf(Consulté le 30/01/2017)

Real-Time Linux Wiki **In:** *RT PREEMPT HOWTO, About the RT-Preempt patch*, https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO#About_the_RT-Preempt_Patch(Consulté le 30/01/2017)

Xenomai Official Website, *How does Xenomai deliver real-time?* **[en ligne].** Disponible sur : https://xenomai.org/start-here/#How_does_Xenomai_deliver_real-time (Consulté le 25/01/2017)

GERUM, Philippe, Xenomai 3 : Hybride et caméléon. **In:** *Open Silicium n°016* **[en ligne]** Mis en ligne en Octobre 2015. Disponible sur : http://connect.ed-diamond.com/Open-Silicium/OS-016/Xenomai-3-hybride-et-cameleon (Consulté le 25/01/2017)

BROWN, Jeremy H., MARTIN, Brad, *How fast is fast enough? Choosing between Xenomai and Linux for real-time applications* **[en ligne].** Disponible sur : https://www.osadl.org/fileadmin/dam/rtlws/12/Brown.pdf (Consulté le 25/01/2017)

Xenomai Official Website, Introducing Xenomai 3 **[en ligne]** Disponible sur : https://xenomai.org/introducing-xenomai-3/ (Consulté le 25/01/2017)

CHABAL, David, Premiers pas avec Xenomai **In:** *Developpez.com* **[en ligne].** Mis en ligne le 12/02/2012. Disponible sur : http://dchabal.developpez.com/tutoriels/linux/xenomai/ (Consulté le 25/01/2017)

Xenomai Official Website, Xenomai Documentation **[en ligne].** Disponible sur : https://xenomai.org/documentation/xenomai-3/html/xeno3prm/index.html (Consulté le 25/01/2017)

BLAESS, Christophe, Raspberry Pi et Temps Réel **In:** *GNU/Linux Magazine HS n° 075* **[en ligne].** Mis en ligne en Novembre 2014. Disponible sur : http://connect.ed-diamond.com/GNU-Linux-Magazine/GLMFHS-075/Raspberry-Pi-et-temps-reel (Consulté le 25/01/2017)

Raspberry Pi Foundation, Raspberry Pi 3 specifications. **[en ligne].** Disponible sur : https://www.raspberrypi.org/products/raspberry-pi-3-model-b/ (Consulté le 25/01/2017)

GREG, Raspberry Pi 3 GPIO Pin Layout **In:** *MCM Electronics Blog* **[en ligne].** Mis en ligne le 26/02/2016. Disponible sur : http://blog.mcmelectronics.com/post/Raspberry-Pi-3-GPIO-Pin-Layout (Consulté le 25/01/2017)

Raspberry Pi Foundation, Frequently Asked Questions. **[en ligne].** Disponible sur : https://www.raspberrypi.org/help/faqs/#powerReqs (Consulté le 25/01/2017)

BLAESS, Christophe, Xenomai 3 sur Raspberry Pi 2 **In:** *Christophe Blaess, Ingénierie et formations sur les systèmes libres* **[en ligne].** Mis en ligne le 22/05/2016. Disponible sur : http://www.blaess.fr/christophe/2016/05/22/xenomai-3-sur-raspberry-pi-2/ (Consulté le 25/01/2017)