

**Rapport de projet DE Stijl 2.0**

21/02/2022

**4 IR A1**

**EMERY Martin** (conception, …)

**PONTALIER Marie** (conception, …)

**TOMIETTO Vincent** (conception, …

Table des matières

[**1.** **Conception** 3](#_Toc96270610)

[**1.1** **Diagramme fonctionnel général** 3](#_Toc96270611)

[**1.2** **Groupe de threads gestion du moniteur** 3](#_Toc96270612)

[**1.2.1** **Diagramme fonctionnel du groupe gestion moniteur** 3](#_Toc96270613)

[**1.2.2** **Description des threads du groupe gestion moniteur** 3](#_Toc96270614)

[**1.2.3** **Diagrammes d’activités du groupe gestion moniteur** 3](#_Toc96270615)

[**1.3** **Groupe de threads gestion du robot** 3](#_Toc96270616)

[**1.3.1** **Diagramme fonctionnel du groupe gestion robot** 3](#_Toc96270617)

[**1.3.2** **Description des threads du groupe gestion robot** 3](#_Toc96270618)

[**1.3.3** **Diagrammes d’activité du groupe gestion robot** 3](#_Toc96270619)

[**1.4** **Groupe de threads vision** 3](#_Toc96270620)

[**1.4.1** **Diagramme fonctionnel du groupe vision** 3](#_Toc96270621)

[**1.4.2** **Description des threads du groupe vision** 3](#_Toc96270622)

[**1.4.3** **Diagrammes d’activité du groupe vision** 3](#_Toc96270623)

[**2.** **Transformation AADL vers Xenomai** 4](#_Toc96270624)

[**2.1** **Thread** 4](#_Toc96270625)

[**2.1.1** **Instanciation et démarrage** 4](#_Toc96270626)

[**2.1.2** **Code à exécuter** 4](#_Toc96270627)

[**2.1.3** **Niveau de priorités** 4](#_Toc96270628)

[**2.1.4** **Activation périodique** 4](#_Toc96270629)

[**3.1** **Donnée partagée** 5](#_Toc96270630)

[**3.1.1** **Instanciation** 5](#_Toc96270631)

[**3.1.2** **Accès en lecture et écriture** 5](#_Toc96270632)

[**3.2** **Port d’événement** 5](#_Toc96270633)

[**3.2.1** **Instanciation** 6](#_Toc96270634)

[**3.2.2** **Envoi d’un événement** 6](#_Toc96270635)

[**3.2.3** **Réception d’un événement** 6](#_Toc96270636)

[**3.3** **Ports d’événement-données** 6](#_Toc96270637)

[**3.3.1** **Instanciation** 6](#_Toc96270638)

[**3.3.2** **Envoi d’une donnée** 7](#_Toc96270639)

[**3.3.3** **Réception d’une donnée** 7](#_Toc96270640)

[**3.** **Analyse et validation de la conception** 7](#_Toc96270641)

1. **Conception**
   1. **Diagramme fonctionnel général**
   2. **Groupe de threads gestion du moniteur**
      1. **Diagramme fonctionnel du groupe gestion moniteur**
      2. **Description des threads du groupe gestion moniteur**
      3. **Diagrammes d’activités du groupe gestion moniteur**
   3. **Groupe de threads gestion du robot**
      1. **Diagramme fonctionnel du groupe gestion robot**
      2. **Description des threads du groupe gestion robot**
      3. **Diagrammes d’activité du groupe gestion robot**
   4. **Groupe de threads vision**
      1. **Diagramme fonctionnel du groupe vision**
      2. **Description des threads du groupe vision**
      3. **Diagrammes d’activité du groupe vision**
2. **Transformation AADL vers Xenomai**
   1. **Thread**
      1. **Instanciation et démarrage**

Chaque thread a été implémenté par un RT\_TASK déclarés dans le fichier **tasks.h** (voir Figure 1). La création de la tâche se fait à l’aide du service **rt\_task\_create** (voir Figure 2) et son démarrage à l’aide de **t\_task\_start** (voir Figure 3). Toutes les tâches sont créées dans la méthode **Init** de **tasks.cpp** et démarrées dans la méthode **Run**. La création et le lancement d’une tâche sont surveillés et en cas d’échec nous sortons du programme.



Figure : déclaration de la tâche **server** dans **tasks.h**

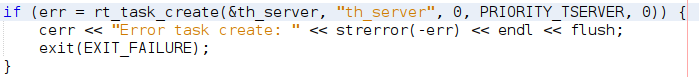


Figure : création de la tâche **server** dans **tasks.cpp**

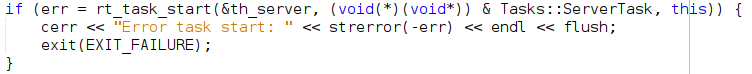


Figure : démarrage de la tâche **server** dans **tasks.cpp**

* + 1. **Code à exécuter**

Lors du lancement d’une tâche avec **rt\_task\_start**, on précise la tâche et l’adresse de la routine qui va être exécutée par cette tâche (voir Figure 2).

* + 1. **Niveau de priorités**

Sous Xenomai le niveau de priorité d’un thread peut aller du 0 à 99, avec 0 la priorité la plus faible. Cette priorité est fixée lors de l’appel au service **rt\_task\_create**. Pour faciliter la modification de ces priorités, nous avons utilisé des #define de la forme **PRIORITY\_TTASK** dans le fichier **tasks.cpp** (voir Figure 4).



Figure : exemple de définition de priorité pour la tâche **server**

* + 1. **Activation périodique**

Pour rendre périodique l’activation d’un thread, on fait appel aux services **rt\_task\_set\_periodic** pour in Pour rendre périodique l’activation d’un thread, on fait appel aux services **rt\_task\_set\_periodic** pour indiquer la période d’une tâche puis au service **rt\_task\_wait\_period** pour descheduler la tâche en attente du cycle suivant (voir Figure 5). Ces services sont appelés après le lancement de la tâche concernée.

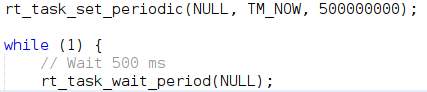


Figure : exemple d'activation périodique d'une tâche

* 1. **Donnée partagée**
     1. **Instanciation**

Les données partagées sont définies comme variables globales dans **tasks.h** et instanciées dans **tasks.cpp**.

* + 1. **Accès en lecture et écriture**

Toutes les variables globales partagées sont protégées par des mutex **RT\_MUTEX**, déclarés dans **tasks.h** (voir Figure 6) et instanciés dans **tasks.cpp** de façon analogue aux tâches (voir Figure 7). Pour chaque modification d’une variable globale, le mutex associés est pris via **rt\_mutex\_acquire** puis relâché via **rt\_mutex\_release** (voir Figure 8). Ainsi on évite tout problème de cohérence et d’accès concurrents sur les données partagées.



Figure : déclaration d'un mutex dans **tasks.h**

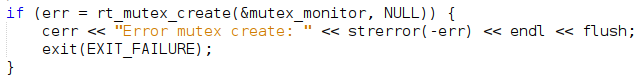


Figure : instanciation d'un mutex dans **tasks.cpp**

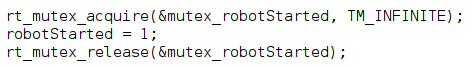


Figure : exemple d’utilisation d'un mutex pour protéger la donnée partagée **robotStarted**

* 1. **Port d’événement**

Pour synchroniser nos tâches via des évènements, nous faisons appel à des sémaphores et des queues. Les sémaphores permettent de se synchroniser via des mécanismes d’incrémentation et de décrémentation (p, v et broadcast).

* + 1. **Instanciation**

Les sémaphores sont déclarés dans le **tasks.h** (voir Figure 9). Ils sont instanciés dans **tasks.cpp** de façon analogue à la création des tâches (voir Figure 10).



Figure : déclaration d'un sémaphore dans **tasks.h**

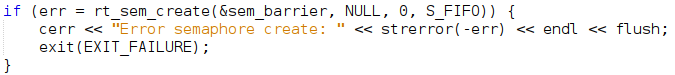


Figure : instanciation d'un sémaphore

* + 1. **Envoi d’un événement**

Pour permettre l’activation d’une tâche en l’attente d’un événement, on incrémente le sémaphore correspondant via le service **rt\_sem\_v** (voir Figure 11. Il est aussi possible d’utiliser le service **rt\_sem\_broadcast** si plusieurs tâches se synchronisent via le même événement (voir Figure 12).

****

Figure : incrémentation d'un sémaphore

****

Figure : broadcast d'un sémaphore

* + 1. **Réception d’un événement**

Une tâche en attente d’un événement est bloquée jusqu’à l’occurrence de cet événement. En pratique, le service **rt\_sem\_p** est bloquant tant que le sémaphore en argument vaut 0. Quand l’événement a lieu et que le sémaphore est modifié (voir 3.2.2), ce service décrémente le sémaphore et le code de la tâche situé après le service peut s’exécuter. Dans une boucle infinie, ce qui est le cas de nos tâches, après exécution du code la tâche se remet ainsi en attente de l’événement à chaque boucle (voir Figure 13).

****

Figure : décrémentation d'un sémaphore

* 1. **Ports d’événement-données**

Les ports d’événements-données sont à la fois des ports d’événements (déclenchement d’actions dans des tâches à la suite d’événements, comme dans le 3.2) et de données (des données sont en même temps transmises à ces tâches). De simples sémaphores ne sont ainsi plus adaptés. Nous utilisons donc ici des files de messages.

* + 1. **Instanciation**

Sous Xenomai, nous allons utiliser des RT\_QUEUES, déclarées dans **tasks.h** (voir Figure 14) et instanciées dans **tasks.cpp** via le service **rt\_queue\_create** qui prend en paramètres un pointeur sur la file de messages, son nom, sa taille, le nombre maximum de messages dans la file et un mode (voir Figure 15).



Figure : déclaration d'une file de messages dans **tasks.h**

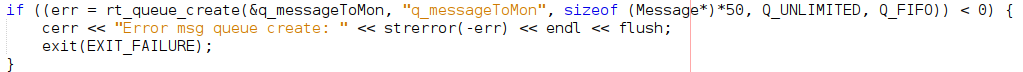


Figure : instanciation d'une file de messages dans **tasks.cpp**

* + 1. **Envoi d’une donnée**

Pour envoyer des données et lancer les tâches en attente d’éléments sur ces files, nous utilisons la tâche **WriteInQueue** déjà implémentée. Elle permet de mettre des messages dans une queue (voir Figure 16).



Figure : exemple d'utilisation de **WrinteInQueue** pour envoyer un message **msgSend** dans la queue **q\_messageToMon**

* + 1. **Réception d’une donnée**

Les tâches en attente d’une donnée pour poursuivre leur exécution ont un comportement similaire à celui d’une tâche en attente sur un sémaphore. En effet, la tâche **ReadInQueue** utilisée pour lire un message est bloquante tant qu’aucun élément n’est disponible dans la file. Lorsqu’un élément est disponible, la tâche est débloquée et lit la donnée en la retirant de la queue (voir Figure 17).



Figure : exemple d'utilisation de **ReadInQueue** pour récupérer un message dans la file et le stocker dans **msg**

1. **Analyse et validation de la conception**