Architectures matérielle et logicielle du robot Omni

Rapport technique n^o. 02082

Geovany Araújo Borges
Département de Robotique
LIRMM, UMR CNRS/Université Montpellier II, n^o. C55060
161, rue ADA. 34392 - Montpellier - Cedex 5 - France

Résumé

Le robot Omni est une plate-forme expérimentale utilisée dans l'évaluation des travaux de recherche sur le suivi de trajectoires et sur la localisation et cartographie simultanées, développés au sein du Département de Robotique du LIRMM. Dans ce document, nous présentons de façon synthétique les architectures matérielle et logicielle du robot Omni.

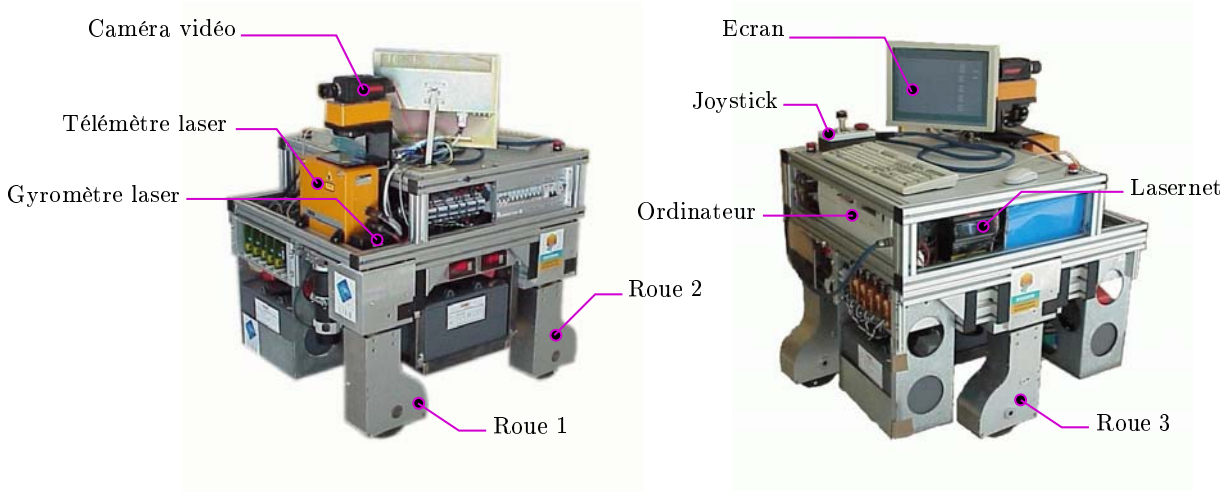
1 Introduction

La plate-forme mobile Omni, illustrée sur la Figure 1, a été conçue pendant la thèse de Le Corre [1][2], au LIRMM. Actuellement, les travaux en cours sur cette plate-forme concernent la commande de ce robot et des approches de localisation et de cartographie de l'environnement. Au niveau de la commande, les algorithmes développés permettent au robot de suivre une trajectoire définie par une courbe de Bézier en utilisant des points de contrôle [3][4]. Cette méthode permet de considérer des estimées incertaines de position absolue du robot et de corriger la trajectoire tout en préservant la continuité du mouvement. Son intégration avec un système de localisation autonome [5] permet aujourd'hui au robot Omni de se déplacer dans des environnements cartographiés.

Omni est un robot omnidirectionnel, doté de trois roues motrices et directrices, qui lui permettent de naviguer facilement dans des environnements encombrés. Son architecture informatique est composée d'un ordinateur central Pentium II 300 MHz, et de cartes *transputer* qui servent d'interface entre l'ordinateur et certains capteurs. Son architecture logicielle a été développée sous *Windows NT*[1], complété de l'extension temps réel *RTX*[2]. Cette plate-forme est dotée des capteurs suivants :

- **Capteurs proprioceptifs** :
  - trois codeurs absolus, utilisés pour mesurer les angles de direction de chaque roue ;
  - six codeurs incrémentaux, utilisés pour mesurer les rotations de chacun des axes ;
  - un gyromètre laser à fibre optique, qui fournit des mesures précises de variation du cap.

[1]Windows NT est une marque enregistrée de Microsoft, Inc.
[2]RTX est une marque enregistrée de Venturcom, Inc.



(a) (b)

FIG. 1 – Robot Omni.

- **Capteurs extéroceptifs** :
  - une caméra vidéo en noir et blanc ;
  - un télémètre laser à balayage horizontal, pour capturer des images de profondeur 2-D de l'environnement ;
  - un goniomètre laser (lasernet), utilisé pour la détection de balises artificielles codées [6]. Cet instrument n'a pas été utilisé dans notre étude.

Ce rapport présente de façon concise les architectures matérielle et logicielle du robot Omni, respectivement dans les sections 2 et 3. Il donne un aperçu général du robot, et met en évidence les points les plus importants de ce système. Cependant, notre objectif n'est pas de donner des schémas de circuits et des listings de programmation.

2 Architecture matérielle

Nous avons sur la Figure 2 le schéma de l'architecture matérielle du robot Omni. Cette architecture, relie les différents modules capteurs et actionneurs à l'ordinateur central du robot. Cette machine est connectée aux différents capteurs par le biais des bus PCI et ISA. Des périphériques permettent l'interaction entre l'utilisateur et Omni : un écran à cristaux liquides, un clavier, une souris et une manette.

Connectées au bus PCI, nous avons une carte Matrox pour l'acquisition d'images et une carte entrées/sorties développée par le service technique du LIRMM. Les données vidéo sont transférées directement dans la mémoire physique de l'ordinateur central par DMA (*Direct Memory Access*). Sur le bus PCI version 2.1 (32 bits/33 MHz), la vitesse de transfert par DMA peut atteindre 132 Mo/s. La carte entrées/sorties permet de lire l'état de la manette, les données codeur, et d'envoyer des consignes de tension sur les moteurs des axes en utilisant

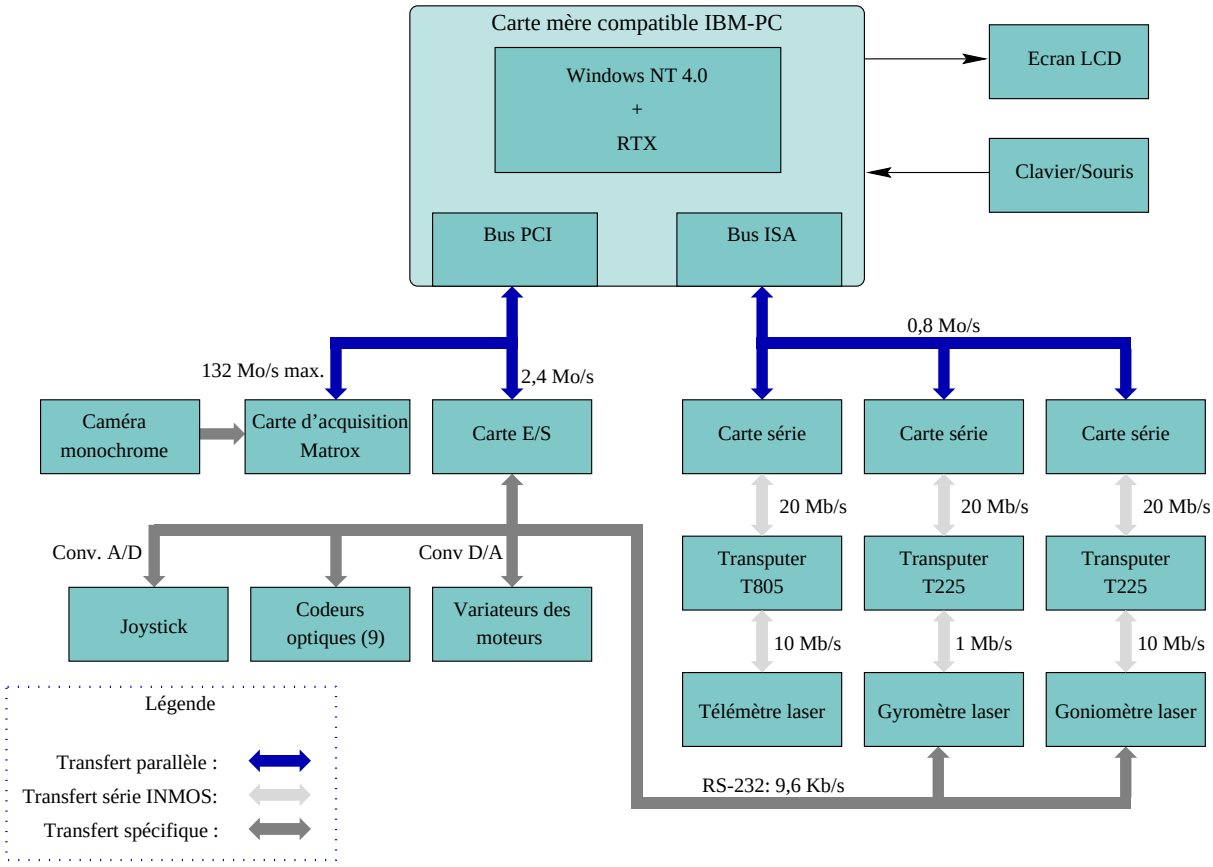


FIG. 2 – Architecture matérielle du robot Omni.

des variateurs. En plus, deux liens série asynchrones permettent de configurer le gyromètre et le goniomètre.

Le bus ISA est utilisé pour connecter trois cartes de communication série *INMOS*, qui servent à communiquer avec des cartes *transputer*. Les cartes *transputer* réalisent l'acquisition de données du télémètre laser, du gyromètre et du goniomètre. L'utilisation des cartes *transputer* est un héritage des architectures plus anciennes développées par le service technique, qui a permis une intégration plus rapide sur Omni, des logiciels des gestion des capteurs étaient déjà opérationnels. Dans un futur proche, il aura intérêt à retirer les cartes *transputer* connectées au gyromètre et au goniomètre, et à connecter ces capteurs directement sur la carte entrées/sorties. Cependant, pour le cas du télémètre laser, le remplacement de sa carte *transputer* n'est pas envisageable parce que le fabricant n'a pas fourni d'information sur son protocole de communication.

3 Architecture logicielle

L'architecture logicielle implantée sur le robot Omni réalise les fonctions suivantes :

- Acquisition des données capteur, segmentation, cartographie locale, localisation et cartographie globale, décrites dans [5] ;

- Gestion des données capteurs pour la réalisation des procédures ci-dessus, pour l'affichage et pour l'enregistrement sur disque, pour traitement hors-ligne sous MATLAB ;
- Intégration de différents modes de navigation :
  - Le mode *manuel*, qui permet à l'utilisateur de commander le robot via la manette ;
  - Le mode *automatique*, qui permet à l'utilisateur de définir une trajectoire qui doit être suivie par le robot ;
  - Le mode *mission*, où trajectoire est chargée sous forme de fichier.
- Construction en ligne de cartes d'environnement, ou chargement à partir d'un fichier.

Les modes *mission* et *automatique* sont basés sur les travaux effectués par Julien Aragones [3] pour la correction de trajectoire. Pour ces modes, la localisation joue un rôle important, étant donné que l'estimée de position est utilisée systématiquement dans le calcul de correction de trajectoire. Malgré l'intérêt dans ce type de système, des modules de détection de défaillances et d'autolocalisation n'ont pas été pour l'instant implantés. On remarque que, étant donné les objectifs fixés pour l'évaluation des méthodes de localisation et de cartographie, l'intégration d'algorithmes de planification de trajectoires n'était pas nécessaire. Mais cela n'implique pas que l'architecture logicielle soit moins complexe.

Nous commençons la présentation de l'architecture logicielle du robot Omni par une brève synthèse sur l'exécution de tâches temps réel sous l'ensemble *WinNT* 4.0/*RTX* 4.3, dans la section 3.1. Ensuite, nous abordons dans la section 3.2 l'organisation des processus de l'architecture. Les structures particulières de stockage temporaire de données, les queues circulaires, sont présentées dans la section 3.3.

3.1 Caractéristiques de l'ensemble *WinNT* 4.0/*RTX* 4.3

Windows NT (ou *WinNT*) est un système d'exploitation générique multitâches/multi-utilisateur qui utilise un ordonnanceur *round-robin* préemptif à priorité dynamique. Cet ordonnanceur cherche à maximiser la densité de tâches actives, au dépit d'un temps de latence important. *RTX* (de *Real-Time eXtension*), comme son propre nom le dit, est une extension temps réel de *WinNT*, qui s'intègre à ce dernier de la façon illustrée sur la Figure 3. En effet, *RTX* est un sous-système qui a son propre noyau, un ordonnanceur préemptif à priorités fixes et structures de contrôle et de synchronisation de tâches [7].

Les caractéristiques principales d'un système *WinNT*/*RTX* sont les suivantes :

- Le noyau du *WinNT* est traité comme une tâche de fond de *RTX*. Il s'exécute toujours quand il n'y a pas de tâche *RTX* active, pour moindre que soit sa priorité. Comme l'interface graphique est complètement contrôlée par *WinNT*, des fonctions d'impression de caractères telles que `printf` deviennent non déterministes, lorsque leur exécution est gérée par *WinNT*. Cependant, *RTX* contient la librairie RTAPI_W32, qui inclut certaines fonctions de l'API (*Application Interface*) de *WinNT* qui ont été réécrites de sorte qu'elles soient déterministes ;
- Une extension de la HAL (*Hardware Abstraction Layer*) permet l'accès direct au matériel. Sans *RTX*, accéder à des dispositifs d'entrée/sortie est beaucoup plus complexe sous *WinNT* ;
- Des opérations en virgule flottante sont autorisées dans des tâches *RTX*, ce qui n'est pas le cas dans des pilotes de dispositifs sous le noyau *WinNT*.

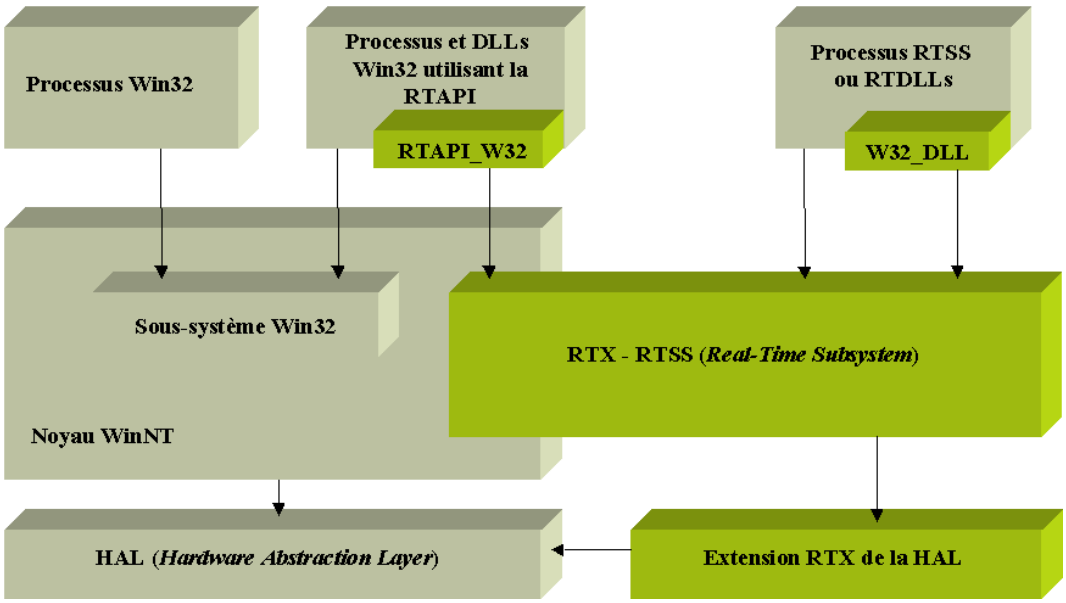


FIG. 3 – Organisation des modules qui composent l'ensemble WinNT - RTX (source : *RTX 4.3 User's Guide, VenturCom Inc., 1999*)

Les fonctionnalités intégrées sur *RTX* sont : des objets de communication interprocessus (*i.e.*, sémaphores, mutex, mémoire partagée), des temporisateurs et horloges, la gestion du bus et des interruptions et l'accès à des fichiers. On remarque cependant que ces sont les outils minimum nécessaires pour la conception de systèmes temps réel. Si nécessaire, d'autres outils tels que les boîtes de messages doivent être conçus et gérés par l'utilisateur. *RTX* ne présente pas de support réseau ; tout accès doit passer par *WinNT*. Cela limite beaucoup son utilisation pour des systèmes commandés à distance et qui ont besoin d'un temps de réponse très court et déterministe.

3.2 Organisation des processus

Pour la mise en œuvre des fonctionnalités de l'architecture logicielle, nous avons divisé la programmation en trois processus. Ces processus et leurs tâches associées sont illustrés sur la Figure 4 :

OmniMonitor : ce processus se charge de la saisie de commandes de la part de l'utilisateur, de l'affichage de l'état général du système (*i.e.*, données capteur, estimées de position, *etc.*) et de l'enregistrement de toutes données capteur sur le disque. Les données enregistrées sur disque peuvent être récupérées sous Matlab pour un traitement hors ligne. Ce processus, géré par le noyau *WinNT*, n'a qu'une seule tâche, *WinMain*, qui s'exécute à une période d'approximativement 500 *ms*. C'est à cette période que la fenêtre utilisateur est rafraîchie. Les données capteur sont enregistrées sur le disque toutes les 1500 *ms* (voir section 3.3 à propos de la façon dont ces données sont stockées) ;

OmniGrab : ce processus s'occupe de l'acquisition d'images vidéo. Ceci est un processus *RTX* géré par le noyau *WinNT*. Il est composé d'une tâche qui se réveille chaque 10 *ms* pour répondre à des requêtes liées à l'acquisition d'images vidéo et à l'affichage de droites verticales

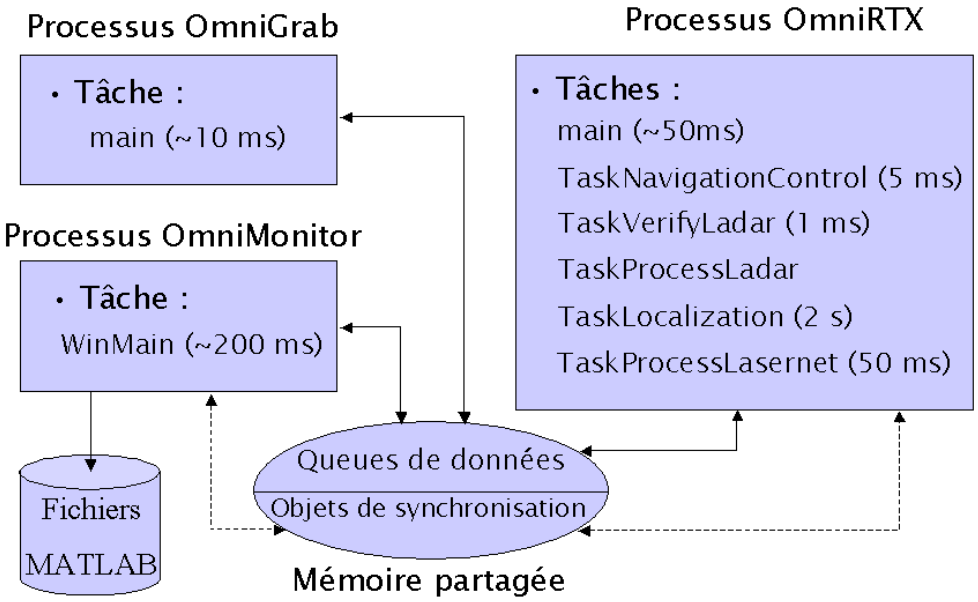


FIG. 4 – Processus logiciels.

résultant de la segmentation de ces images. Ce processus réalise l'acquisition multiple d'images vidéo (*c.f.* [5], section 3.4). Ainsi, tant que l'acquisition d'une image télémétrique n'est pas finie (voir tâche *TaskProcessLadar* du processus *OmniRTX* ci-après), des images vidéo sont acquises, datées et stockées dans la mémoire. L'image vidéo $\mathcal{I}$ présentant le plus petit décalage temporel par rapport à l'image télémétrique $\mathcal{L}$ est ensuite associée à cette dernière pour être traitée par le procédé de localisation ;

OmniRTX : c'est dans ce processus que les tâches soumises à des contraintes temps-réel sont éxecutées, comme par exemple la localisation, la cartographie, l'acquisition de données capteur et le suivi de trajectoire. Ce processus est composé de six tâches :

- la tâche *main*, qui est la première tâche lancée lors de l'initialisation du processus, s'occupe de l'initialisation du système, et de l'exécution de commandes sollicitées par le processus *OmniMonitor*. En effet, *main* est une tâche de fond, de priorité minimale, qui se réveille chaque 50 *ms* pour traiter des requêtes du processus **OmniMonitor** qui se trouvent en mémoire partagée ;
- la tâche *TaskNavigationControl*, qui est celle de plus haute priorité gérée par *RTX*. En effet, cette tâche gère l'exécution des procédures de commande bas niveau des moteurs du robot, de l'acquisition des données du gyromètre laser et des codeurs optiques, et de l'estime. Elle s'exécute à des périodes de 5 *ms* ;
- la tâche *TaskProcessLasernet*, qui interroge le lasernet chaque 50 *ms* pour vérifier si une balise a été détectée. Dans ce cas, les informations sur la balise (*i.e.*, code et position du lasernet par rapport à la balise) sont transférées vers l'ordinateur central ;
- les tâches *TaskVerifyLadar* et *TaskProcessLadar*, gèrent l'acquisition d'images télémétriques. Comme le télémètre laser balaye de façon continue l'environnement, ses images sont toujours envoyées à l'ordinateur central à la fin de leur génération. Pour minimiser le retard lié à l'acquisition de ces images, *TaskVerifyLadar* est une tâche qui s'exécute à

des courtes périodes de 1 *ms* pour interroger le *transputer* du télémètre afin de détecter la présence d'une image prête à être transférée vers l'ordinateur central. Dans ce cas, la tâche *TaskProcessLadar* est lancée par *TaskVerifyLadar* pour s'occuper du transfert de l'image. A la fin du transfert, si une requête a été faite durant son acquisition, l'image est pré-traitée (*i.e.*, correction du biais, compensation de la déformation due au mouvement du capteur) et disponible pour être ensuite utilisée par la tâche de localisation ;

- la tâche *TaskLocalisation*, qui s'exécute chaque 2000 *ms* pour estimer la position absolue du robot par mise en correspondance de cartes géométriques de l'environnement. Cette tâche réalise la séquence de procédures suivantes :

  1. Le lancement de requêtes pour les tâches d'acquisition de données extéroceptives (*i.e.*, images fournies par le télémètre laser et par la caméra vidéo) ;
  2. La procédure de pseudo-synchronisation de la paire d'images ($\mathcal{L}$,$\mathcal{I}$) ;
  3. La segmentation de ($\mathcal{L}$,$\mathcal{I}$) ;
  4. La cartographie locale pour construire la carte locale $\mathcal{M}^L$;
  5. La localisation par mise en correspondance entre la carte globale $\mathcal{M}^G$ et la carte locale $\mathcal{M}^L$;
  6. Le recalage temporel de l'estimée de position, étant donné le temps écoulé entre l'acquisition de la paire d'images ($\mathcal{L}$,$\mathcal{I}$) et la localisation par mise en correspondance de cartes ;
  7. Au cours de l'exécution du mode cartographie de l'environnement, la carte globale $\mathcal{M}^G$ est mise à jour ;
  8. Le stockage de $\mathcal{L}$ et de $\mathcal{I}$ dans les queues circulaires respectives (*c.f.* section 3.3).

Les fenêtres utilisateur correspondant aux processus sont montrées sur la Figure 5. Parmi ces fenêtres, la seule qui permet la configuration du système de la part de l'utilisateur est celle qui correspond à **OmniMonitor**. Les autres fenêtres affichent des informations génériques sur l'état des processus.

Les processus ont des dynamiques d'exécution différentes, en fonction du noyau auquel ils sont attachés. *OmniMonitor* est un processus *WinNT*, dont les contraintes d'exécution temps réel ne sont pas très fortes. C'est pour cela que ce processus gère l'interface utilisateur. La création du processus *OmniGrab* est liée au fait que l'acquisition d'images ne peut pas être réalisée au niveau du noyau *RTX*. En effet, Matrox, le fabricant de la carte d'acquisition d'images, ne fournit pas de pilote *RTX* pour ce dispositif. Etant conçu comme un processus *RTX* contrôlé par le noyau du *WinNT*, il a le privilège de s'exécuter tel qu'un *device driver WinNT*, et cela tout en ayant le droit de réaliser du calcul en virgule flottante. En plus, pour minimiser le retard de la commutation de tâches effectuée par le noyau *WinNT*, nous avons associé à cette tâche la priorité maximale admissible par le système. Ainsi, nous pouvons lancer une acquisition d'images avec un retard minimum.

Nous remarquons aussi sur la Figure 4 que la communication entre ces processus se réalise par mémoire partagée. Cela est contrôlé par des objets de synchronisation, qui sont en fait des outils de base de systèmes temps-réel [8]. Ceux-ci ne sont pas décrits dans ce rapport, étant donné leur généralité. Seules les queues circulaires sont présentées dans la section 3.3.

Les tâches ayant des contraintes d'exécution temps réel ont été implémentées dans le processus *OmniRTX*. Le Tableau 1 présente leurs priorités et leurs temps d'exécution (T_{CPU})

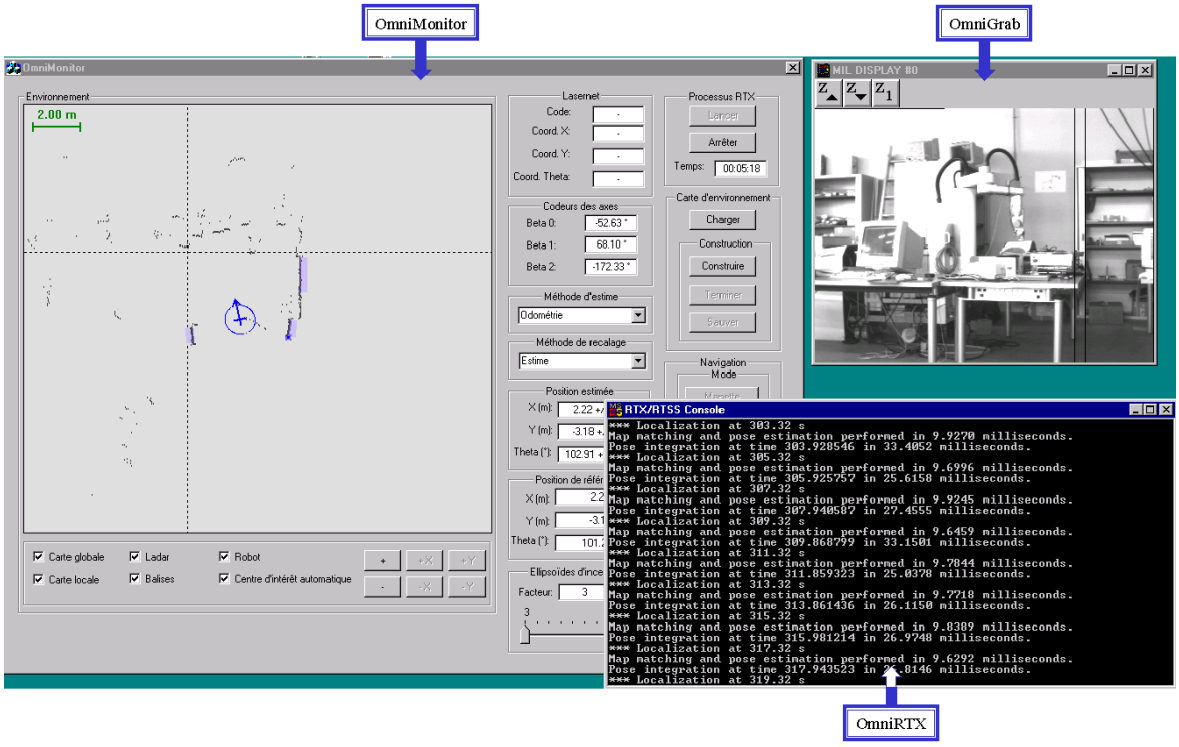


FIG. 5 – Fenêtres utilisateur des processus du robot Omni

minimum et maximum lors d'une expérience avec le robot qui a duré une dizaine de minutes. Dans cette expérience, les différentes fonctionnalités de l'architecture logicielle ont été activées. Sur *RTX*, les priorités des tâches peuvent aller de `RT_PRIORITY_MIN` jusqu'à `RT_PRIORITY_MAX`, ce dernier correspondant à la priorité la plus importante. Dans *OmniRTX*, les priorités sont choisies en fonction de l'importance du traitement réalisé par chaque tâche. La tâche *TaskNavigationControl* est pourtant la plus prioritaire, étant donné le besoin de maintenir stable la périodicité d'exécution du contrôleur de vitesse des roues. Inversement, les tâches *main* et *TaskLocalization* sont les moins prioritaires. En effet, étant donnée leur finalité, des interruptions de l'ordre d'une vingtaine de millisecondes ne sont pas gênantes. Le temps d'exécution de certaines tâches dépend aussi du mode de navigation, et du fait que la mise à jour de la carte globale est autorisée ou non. Le mode de navigation affecte principalement la tâche *main*, qui doit simuler le trajet à réaliser le robot à partir des points de passage crées dans le mode *mission*. Nous remarquons sur le Tableau 1 que cette tâche a présenté un temps d'exécution maximum de l'ordre d'une seconde. Ceci est dû au fait que, dans ce mode, le générateur de mouvement par courbes de Bézier a été appliqué pour simuler un trajet composé de 130 points de passage. Un autre fait intéressant est que la procédure de cartographie découplée ne rajoute pas davantage de temps d'exécution à la tâche *Task-Localisation*. En effet, ce sont les autres procédures de cette tâche qui occupent le plus de temps.

TAB. 1 – Mesures de temps d'exécution des tâches du processus *OmniRTX*.

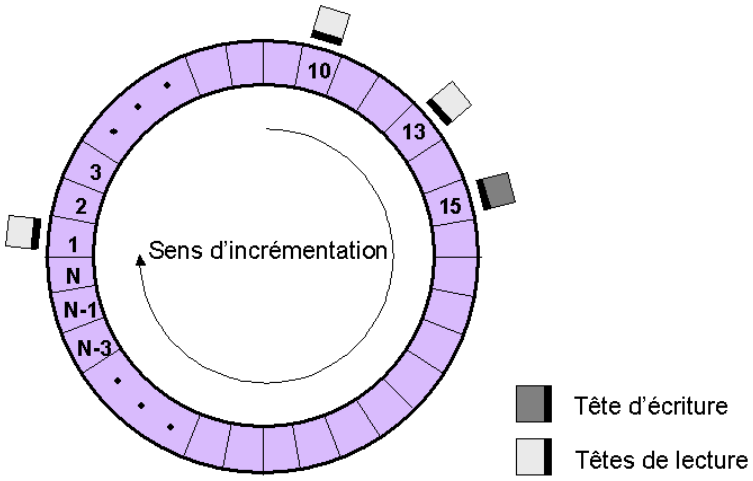
Tâche	Priorité	T_{CPU} min.	T_{CPU} max.
main (mode manuel)	`RT_PRIORITY_MIN`	$0,004\ ms$	$0,766\ ms$
main (mode mission)	`RT_PRIORITY_MIN`	$0,004\ ms$	$1050,631\ ms$
TaskNavigationControl (mode manuel)	`RT_PRIORITY_MAX`	$0,464\ ms$	$0,852\ ms$
TaskNavigationControl (mode mission)	`RT_PRIORITY_MAX`	$0,499\ ms$	$0,911\ ms$
TaskProcessLasernet	`RT_PRIORITY_MAX-3`	$0,005\ ms$	$0,028\ ms$
TaskVerifyLadar	`RT_PRIORITY_MAX-2`	$0,003\ ms$	$0,652\ ms$
TaskProcessLadar	`RT_PRIORITY_MAX-1`	$7,742\ ms$	$14,709\ ms$
TaskLocalisation (sans la cartographie)	`RT_PRIORITY_MIN`	$381,902\ ms$	$705,559\ ms$
TaskLocalisation (avec la cartographie)	`RT_PRIORITY_MIN`	$361,922\ ms$	$811,172\ ms$

3.3 Structures de stockage de données

Le stockage temporaire de toutes les données capteur du robot Omni se fait à l'aide de queues circulaires. Ce type de structure est illustré sur la Figure 6(a). Une queue circulaire est en fait un tableau auquel une structure de contrôle est associée pour gérer l'accès exclusif à ses données (*c.f.* Figure 6(b)). L'accès à la queue se fait via des têtes, dont une tête pour écrire sur le tableau et des têtes multiples pour lire les données. A chaque opération d'écriture ou de lecture, la tête associée se décale dans un sens unique. Sur le robot Omni, nous avons une queue circulaire associée à chaque capteur. Les têtes d'écriture de ces queues sont manipulées par les tâches d'acquisition (sauf pour les données vidéo et télémétriques, qui sont stockées dans la tâche *OmniLocalization*). Les têtes de lecture sont manipulées soit pour l'affichage des données les plus récentes, soit pour l'enregistrement sur le disque des données encore non enregistrées par la tâche *WinMain* du processus **OmniMonitor**. De plus, pour des procédures de traitement séquentiel de données, comme pour le recalage temporel de l'estimée de position, une recherche sur ces queues permet de retrouver l'index d'une donnée acquise à un instant passé. La taille de ces structures est déterminée en fonction de la période moyenne d'acquisition des données, de façon qu'elles contiennent toujours les données acquises durant les cinq dernières secondes. Cette période de cinq secondes est plus que suffisante pour satisfaire aux besoins des tâches qui utilisent les données capteur. Nous avons limité le nombre de têtes de lecture à cinq.

Les opérateurs définis pour ces structures sont les suivants :

- `QUEUE_Init` : Initialise les variables de contrôle de la queue circulaire. Cette fonction reçoit comme paramètres la taille du tableau de données et le nombre de têtes de lecture ;
- `QUEUE_RequestWriteIndex` : Incrémente d'une place la tête d'écriture. Cette fonction ne présente jamais d'échec ;
- `QUEUE_RequestReadIndex` : Incrémente d'une place la *n-ième* tête de lecture. Cette fonction présente un échec si la queue est vide ou si la *n-ième* tête de lecture se trouve sur la tête d'écriture avant d'être incrémentée, sachant qu'elle a été lue auparavant. Ces conditions d'échec signifient qu'aucune donnée nouvelle n'est disponible ;
- `QUEUE_RequestLastReadIndex` : Déplace la *n-ième* tête de lecture à la même position de la tête d'écriture. Les conditions d'échec de cette fonction sont les mêmes que celles de `QUEUE_RequestReadIndex` ;



(a)

```
// Header file for QUEUE_CONTROL data type.
#define MAXSIZE_QUEUE_READ_GATES    5
#define QUEUE_READ_GATE_0           0
#define QUEUE_READ_GATE_1           1
#define QUEUE_READ_GATE_2           2
#define QUEUE_READ_GATE_3           3
#define QUEUE_READ_GATE_4           4

typedef struct{
        int         Size;
        int         ReadIndex[MAXSIZE_QUEUE_READ_GATES];
        int         WriteIndex;
        int         NReaders;
        BOOL        FlagStillNotWritten;
        BOOL        FlagStillNotRead[MAXSIZE_QUEUE_READ_GATES];
        BOOL        FlagOverWrite[MAXSIZE_QUEUE_READ_GATES];
}QUEUE_CONTROL, *PQUEUE_CONTROL;
```

Etiquettes pour les têtes de lecture

Structure de contrôle

(b)

FIG. 6 – (a) Queue circulaire pour le stockage temporaire de données et (b) structure de contrôle associée.

- `QUEUE_GetWriteIndex` : Retourne l'index actuel de la tête d'écriture. Cette fonction présente un échec si la queue est vide.

Nous illustrons l'utilisation de ces opérateurs par les exemples suivants.

Exemple 1 (Affichage de la donnée la plus récente) : Cet exemple est utilisé principalement pour l'affichage de données capteur. L'interface utilisateur du processus **OmniMonitor** utilise la séquence de code suivante pour retrouver l'index de la mesure la plus récente des codeurs de direction des roues du robot :

```
if(QUEUE_RequestLastReadIndex(&ArtEncodersQueue, QUEUE_READ_GATE_TO_DISPLAY_WinNT,
&Index) == TRUE){
        // A suivre, la procédure d'affichage de la donnée d'index Index.
        Afficher(ArtEncoders[Index]);
}
```

Dans cette séquence, `QUEUE_READ_GATE_TO_DISPLAY_WinNT` est une macro qui identifie la

tête de lecture de la queue liée à l'affichage, et `ArtEncodersQueue` est la structure de contrôle. Les données sont stockées dans le tableau `ArtEncoders`.

Exemple 2 (Enregistrement des données sur disque) : Cet exemple illustre le mécanisme utilisé pour garantir que toute donnée acquise est enregistrée sur disque. Pour cela, la séquence de code suivante est exécutée à des périodes de 1500 *ms* dans la tâche *WinMain* de **OmniMonitor** :

```
while(QUEUE_RequestReadIndex(&ArtEncodersQueue, QUEUE_READ_GATE_TO_FILE_WinNT,
&Index) == TRUE){
        // A suivre, la procédure d'enregistrement de la donnée d'index Index.
        Enregistrer(ArtEncoders[Index]) ;
}
```

Dans cette séquence, `QUEUE_READ_GATE_TO_FILE_WinNT` est une macro qui identifie la tête de lecture liée à l'enregistrement sur disque. On remarque que cette séquence de code fait appel à la procédure `QUEUE_RequestReadIndex` tant que la tête de lecture n'est pas alignée avec la tête d'écriture qui est sensée pointer vers la dernière donnée stockée. Ainsi, chaque 1500 *ms*, toutes les données acquises dès la dernière exécution de ce code sont enregistrées.

4 Conclusion

Nous avons présenté de façon synthétique les architectures matérielle et logicielle du robot Omni. L'architecture matérielle présente une forme distribuée pour l'acquisition des données capteurs. Cependant, tous les traitements sont effectués sur l'ordinateur principal du robot. En se servant du support de programmation multitâches constitué par l'ensemble *Windows NT / RTX*, nous avons pu développer l'architecture logicielle décrite dans ce rapport. Nous mettons en évidence deux points forts de ce développement :

1. Une des difficultés de l'utilisation de systèmes commerciaux tels que ceux fournis par Matrox est le manque de librairies adaptées à des systèmes temps réel restrictifs (ou *hard real-time systems*). On remarque l'existence de nombreux fournisseurs de librairies pour *Windows NT*. Ces librairies sont normalement utilisées en systèmes d'imagerie sans fortes contraintes d'exécution temps réel. Le système présenté utilise la faculté du processus temps réel *RTX* de communiquer par mémoire partagée avec des processus *Windows NT*, de sorte à pouvoir utiliser les images acquises dans des tâches déterministes. Dans le cas du robot Omni, même si la fréquence d'acquisition d'images de $0,5$ Hz n'est pas très importante, c'est en fait le besoin de synchroniser leur acquisition avec celle d'une image télémétrique qui impose des contraintes de retard minimum. Ces contraintes semblent être respectées, comme l'illustrent les résultats présentés dans [5] ;
2. Les queues circulaires sont un mécanisme élégant et efficace de stockage temporaire de données capteur. Dans l'architecture de localisation et de cartographie, certaines procédures font un fort usage de ces structures. Par exemple, c'est le cas de la procédure de correction de la déformation d'images télémétriques et de la procédure de recalage temporel de l'estimée de position ([5], chapitres 3 et 6). Ces procédures permettent au robot d'estimer sa position en même temps qu'il se déplace (*i.e.*, localisation dynamique). De plus, les queues circulaires ont permis l'enregistrement complet de toutes les données acquises par les capteurs du robot lors d'expériences d'évaluation. Cela nous a permis

de construire une base de données multicapteur, qui continue à être utilisée pour évaluer hors ligne sous Matlab de nouveaux algorithmes en cours de développement.

Références

[1] Y. le Corre, A. Fournier, and M. Benoit. Performances of a 3 caster wheeled mobile robot : the advantages of over-actuation. In *International Conference on Advances in Vehicle Control and Safety (ACVS)*, 1998.

[2] Y. le Corre. *Conception et commande d'un robot omnidirectionnel.* PhD thesis, LIRMM / Université Montpellier II, Décembre 1998.

[3] J. Aragones and A. Fournier. Commande dynamique d'un robot omni-directionnel redondant. In *Conférence Internationale Francophone d'Automatique (CIFA)*, 2002.

[4] J. Aragones, Geovany A. Borges, and A. Fournier. Accuracy improvement for a redundant vehicle. In *33rd International Symposium on Robotics (ISR 2002)*, 2002.

[5] G. A. Borges. *Cartographie de l'environnement et localisation robuste pour la navigation de robots mobiles.* PhD thesis, Université Montpellier II, LIRMM, 161 rue ADA, 34392, Montpellier, Cedex 5, France, Mai 2002.

[6] M.-J. Aldon and M. Benoit. Localization system of the hospital transport robot FIRST. In *2nd IFAC Conference on Intelligent Autonomous Vehicles*, pages 244–249, Helsinki, Finland, June 12 - 14 1995.

[7] VenturCom, Inc. *RTX 4.3 User's Guide*, 1999.

[8] Philip A. Laplante. *Real-Time Systems Design and Analysis.* IEEE Press, 1997.