TP Temps Réel

Ludovic Saint-Bauzel <saintbauzel@isir.upmc.fr> and Jérôme Pouiller <jezz@sysmic.org>

Polytech Paris UPMC - 2020

Table des matières

1 Avant de commencer

Cette section est purement informative. Il n'y a pas de questions à l'intérieur.

1.1 Documentation

1.1.1 Références des pages de man

Comme le veut l'usage, les références des pages de man sont donnés avec le numéro de section entre parenthèses. Ainsi, *wait(2)* signifie que vous pouvez accéder à la documentation avec la commande `man 2 wait`. Si vous omettez le numéro de section, `man` recherchera la première page portant le nom `wait` (ce qui fonctionnera dans 95% des cas). Le numéro de section vous donne aussi une indication sur le type de documentation que vous aller trouver. D'après *man(1)*, voici les différentes sections :

1. Executable programs or shell commands
2. System calls (functions provided by the kernel)
3. Library calls (functions within program libraries)
4. Special files (usually found in /dev)
5. File formats and conventions (e.g. /etc/passwd)
6. Games
7. Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7)
8. System administration commands (usually only for root)
9. Kernel routines [Non standard]

Vous pouvez trouver une version en ligne de la plupart des pages de man sur The Friendly Manual (https://www.gnu.org/manual/manual.fr.html).

Vous trouverez en particulier les pages relatives à l'API du noyau Linux qui ne sont pas présentes en local sur vos machine.

Prenez néanmoins garde aux éventuelles différences de versions d'API (très rare et quasiment exclusivement sur les pages relatives à l'ABI du noyau).

1.1.2 Standards

Vous subissez aujourd'hui tout le poids de l'histoire de la norme Posix (30 ans d'histoire de l'informatique). Ne soyez pas étonnés de trouver de multiples interfaces pour une même fonctionnalité. C'est particulièrement vrai sur les fonction relatives au temps.

Nous basons ces exercice sur la norme POSIX.1-2001 que vous pourrez trouver sur http://www.unix-systems.org/version3/online.html

Vous trouverez plus d'informations sur l'histoire des différentes normes existantes sur *standards(7)*.

1.2 Erreurs

Pour indiquer quelle erreur s'est produite, la plupart des fonctions Posix utilisent :
- soit leur valeur de retour
- soit une variable globale de type `int` nommée `errno`

Utilisez *strerror(3)* pour obtenir le message d'erreur équivalent au code de retour.

Exemple :

```
1 if (err = pthread_create(&tid, NULL, f, NULL))
    printf("Task create: %s", strerror(err));
  if (-1 == pause())
    printf("Pause: %m");
```

1.3 Compilation

Pour la plupart des exercices, vous aurez besoin de linker [1] avec `-lpthread` et `-lrt`.

Par ailleurs, nous vous conseillons fortement de compiler avec l'option `-Wall` de `gcc` et d'utiliser un système de Makefile.

1. C'est à dire la phase de compilation où vous liez tous vous fichiers `.o` ensemble

2 OROCOS : Développer un contrôleur de moteur

La méthodologie de ce TP consiste à vous focaliser sur des documentations existantes et trouver les informations utiles dans les codes sources et les documents associés. Vous trouverez toutes les informations sur la création et le codage de composants dans le fichier `orocos-components-manual.pdf`. Pour vous accélérer le travail vous avez aussi disponible un fichier contenant les sources de tutoriaux disponibles en ligne pour apprendre à programmer avec OROCOS (`rtt-exercises-2.7.0.tar.gz`).

Question 2.1. *On souhaite réaliser un programme qui fasse un asservissement en orientation d'un moteur. Dessinez le contrôleur qui répond au besoin.*

Documentation utile: *SADT*

Question 2.2. *Dans un premier temps en lisant le document* `orocos-components-manual.pdf` *trouvez le squelette minimal d'un composant.*

Question 2.3. *Essayez de compiler ce squelette. Pour cela il faut tout d'abord créer un workspace comme expliqué dans l'appendice A, puis dans le dossier* `src` *de votre workspace.*

Le squelette du code à compiler peut-être créé en utilisant la commande :
`orocreate-catkin-pkg MyTask component`
et en modifiant les sources de `mytask-component.cpp` *et de* `mytask-component.hpp` *avec le squelette que vous avez trouvé à la question précédente. Compilez et chargez le dans le* `deployer`

Documentation utile: *deployer-gnulinux, orocreate-catkin-pkg*

Question 2.4. *Vous êtes maintenant en mesure de compiler et d'executer un composant.*

Modifiez ce squelette pour qu'il implémente les fonctions de base :
- *updateHook*
- *configureHook*
- *startHook*
- *stopHook*

Et que chacune écrive dans les log d'OROCOS.

```
RTT::log(RTT::Info) << "Text that goes in the log file !" <<RTT::endlog();
```

Documentation utile: *RTT : :log, updateHook, configureHook, startHook, stopHook*

Question 2.5. *Dans ce composant on va ajouter : un port en entrée (mesure), un port en sortie (cmd) et un attribut publique de valeur désirée (des).*

Documentation utile: *inputPort<>, outputPort<>, ports()->addPort(),addAttribute()*

Question 2.6. *Il est possible de faire plusieurs composants dans les mêmes fichiers sources (cf. hello-3-dataports). Pour cela, il faut utiliser une macro différente à la fin du fichier* `*.cpp`.

Ainsi on peut vérifier que cela fonctionne. Ensuite il suffira d'ajouter `ORO_LIST_COMPONENT_TYPE(XXX)` *pour chaque composant XXX ajouté à ces sources.*

Question 2.7. *Faisons un déploiement automatisé qui charge notre composant. On peut créer un fichier* `*.ops`. *par exemple* `start.ops`. *On peut ainsi configurer le composant avec une période de 1 seconde.*

Documentation utile: *setPeriod, import, start, loadComponent*

Question 2.8. *Étendons notre code qui présente 1 composant en un code qui implémente les composants suffisants pour réaliser le contrôleur décrit dans la question 2.1. Faisons aussi un déploiement automatisé qui inclut ces composants.*

Question 2.9. *Enfin, nous souhaitons faire en sorte que vos messages soient visible dans ros. Pour cela, on doit utiliser des types compréhensible par ROS. On inclut donc les dépendances avec les paquets qui font le lien.*

Documentation utile: *std_msgs : :Float64(ros), package.xml(ros), rtt(ros), rtt_roscomm(ros), rtt_rosnode(ros), rtt_std_msgs(ros), stream(rtt)*

A Initialisation de ROS-OROCOS

Tout d'abord il faut declarer les variables qui permettent d'acceder aux fonctionnalités de ROS :

```
source /opt/ros/kinetic/setup.bash
```

Dans un premier temps on doit créer un workspace. Pour cela on commence par créer un dossier rosws qui contient un dossier src.

```
mkdir -p rosws/src
```

Ensuite, on entre dans le dossier et on utilise la commande rosws.

```
cd rosws/src
catkin_init_workspace
cd ..
catkin_make
```

Ensuite on ajoute le workspace dans les variable de ros.

```
source devel/setup.bash
```

Enfin on souhaite que ROS fonctionne toujours c'est pourquoi on va modifier le fichier .bashrc pour qu'il charge ros et les workspace.

```
source /opt/ros/kinetic/setup.bash
source ~/rosws/devel/setup.bash
```

B Modalités de rendu

Votre rendu sera effectué sur la plateforme Moodle (https://moodle-sciences.upmc.fr/moodle-2019).

Vous devez rendre votre travail dans une archive gzippée. L'archive devra se nommer `TR_ROB5-nom.tar.gz` et se décompresser dans le répertoire du même nom.

Votre archives ne doit pas contenir de fichiers binaires. Il ne doit contenir que vos sources (fichiers sources, fichiers entêtes, Makefile, etc...).

Exemple de création de rendu :

```
make clean
cd ..
tar cvzf TR_ROB5-pouiller.tar.gz TR_ROB5-pouiller
```

Vous devrez envoyer déposer une version **à la fin de la séance** et vous pouvez faire évoluer ce document jusqu'au début du prochain TP.