

2021/2022

***E*lectronique et *T*echnologies *N*umériques**

**Exécutifs temps-réel
Documentation pour Travaux Pratiques**

O. PASQUIER

– 4^e année –

Exécutifs Temps-Réel

Travaux pratiques

O. Pasquier, Février 2022

Les travaux pratiques ont pour objectif d'illustrer les mécanismes de base des exécutifs temps-réel et les situations les plus couramment rencontrées dans les applications. Ils sont basés sur l'utilisation des produits Workbench pour le développement et VxWorks comme Exécutif Temps-Réel pour la cible. Ces 2 produits sont émis par la société WindRiver.

Les séances de travaux pratiques de temps-réel doivent être considérées comme la combinaison de travaux dirigés et de travaux pratiques, l'objectif pour l'étudiant étant de pouvoir vérifier par la pratique les analyses théoriques qui ont pu être effectuer sur les problèmes posés et lors des TD.

Le sujet du TP1 montre à la fois l'aspect théorique que l'on cherche à mettre en œuvre autour de la synchronisation et de l'exclusion mutuelle ainsi que les différentes vérifications qui ont pour objet de valider la théorie. Ce travail de validation et de recherche des différents tests à effectuer représentent le travail d'un ingénieur. Pour cette raison, le sujet des TP2 et TP3 ne montre pas cet aspect et c'est aux étudiants de déterminer les validations et preuves à mettre en œuvre lors des TP et de montrer cet aspect lors d'un compte-rendu de TP qui porte uniquement sur les TP2 ou TP3. Il est évident que vous pouvez, à titre d'exercice mais aussi pour assoir vos compétences (donc préparer l'évaluation), rédiger un compte-rendu pour le TP1 et le TP pour lequel le compte rendu n'est pas requis.

A la suite du texte des trois manipulations se trouvent

- une aide à l'utilisation de Workbench,
- un extrait de la documentation de VxWorks et concernant seulement les mécanismes utilisés en TP.

Manip 1 : Synchronisation, Exclusion mutuelle

Cette manipulation a pour objectif :

- d'appréhender la mise en oeuvre d'un exécutif temps réel,
- d'analyser la synchronisation par sémaphore,
- d'analyser l'influence des priorités respectives des tâches,
- de mettre en oeuvre l'exclusion mutuelle.

1. SITUATION

On désire implanter la structure fonctionnelle ci-dessous à l'aide d'un exécutif temps-réel

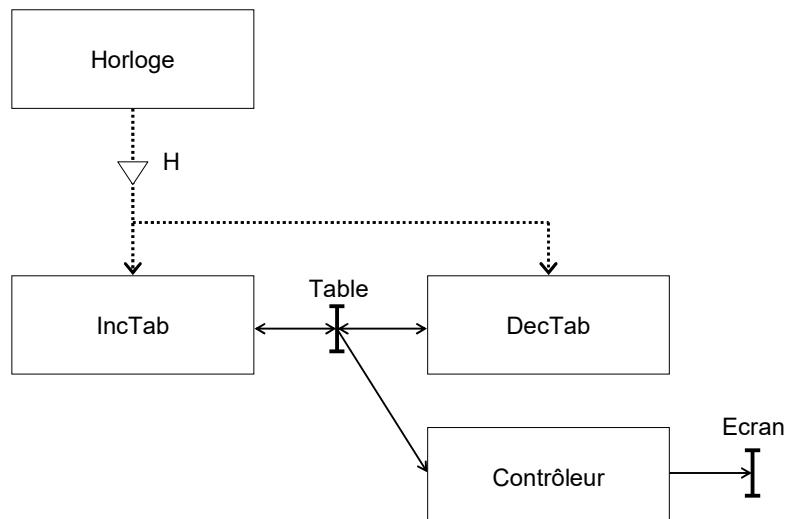


Figure 1 - Structure Fonctionnelle pour Manip 1.

La description comportementale des fonctions est présentée ci-dessous. Table est un tableau de valeurs de type *int* qui peuvent représenter un signal, une rampe par exemple (pensez bien à l'initialiser !!!). Sa taille est une constante (*sizeTable*) qui pourra être modifiée au cours des tests. Lors de l'initialisation, elle prend les valeurs : [0,1,2,..., *sizeTable*-1]

1.1. Fonction Horloge

Il s'agit d'une tâche qui effectue un *taskDelay* et retransmet la synchronisation par l'intermédiaire de *H*.

1.2. Fonction IncTab

Action temporaire activée par *H*. Toutes les 5 activations, la fonction incrémente de 10 la valeur de chaque élément de Table.

Son comportement fonctionnel est le suivant :

```

Action IncTab sur événement H
    avec(entrée/sortie Var Table:array[0..sizeTable-1] of integer);
    var indice : integer;
    var NbActivations : integer;
begin
    NbActivations := 0;
    cycle H:
        if (NbActivations = 0)
            then begin
                for indice:=0 To (SizeTable-1)
                    Table[indice]:= Table[indice]+10;
                NbActivations := 4;
                end;
            else NbActivations := NbActivations - 1;
            end;
        end_cycle;
end_IncTab;

```

1.3. Fonction DecTab

Son comportement fonctionnel est décrit par l'algorithme suivant :

```

Action DecTab sur événement H
    avec(entrée/sortie Var Table:array[0..sizeTable-1] of integer);
    var indice : integer;
begin
    cycle H:
        begin
            for indice:=0 To (SizeTable-1)
                Table[indice]:= Table[indice]-1;
            end;
        end_cycle;
end_DecTab;

```

1.4. Fonction Contrôleur

Il s'agit d'une fonction permanente (tâche de fond) qui vérifie en permanence la cohérence de *Table* et affiche un message d'erreur lorsqu'une incohérence est trouvée dans la *Table*. Pour cela, Elle prend en référence la première valeur de la *Table* (*Table[0]*) et vérifie chaque élément de *Table* par rapport à sa référence (*Table[indice]* = *référence + indice*). Lorsque la *Table* a été totalement parcourue, une nouvelle référence est prise et la *Table* est de nouveau vérifiée.

Son comportement fonctionnel est donc décrit par l'algorithme suivant :

```

Action Contrôleur
    avec(entrée/sortie Var Table:array[0..sizeTable-1] of integer);
    var indice, référence : integer;
    var Erreur : boolean;
begin
    cycle :
        begin
            référence :=Table[0];
            Erreur := False
            for indice:=0 To (SizeTable-1)
                if Table[indice] <> référence + indice then Erreur := True;
                if Erreur then print(...);
            end;
        end_cycle;

```

```
end_Controleur;
```

2. ANALYSE THEORIQUE

Avant le TP, il s'agit de déterminer l'implantation de l'application et d'analyser les différentes situations que vous devez obtenir. La démarche d'un ingénieur est de prédire le comportement théorique et surtout, en cas de désaccord, entre la théorie et les résultats pratiques, être en mesure de déterminer l'erreur d'analyse.

En vous inspirant du fichier modèle.c et en vous limitant dans un premier temps aux fonctions *Horloge*, *DecTab* et *IncTab* ainsi que les relations *H* et *Table* (sans exclusion mutuelle) :

- Déterminer les éléments VxWorks nécessaires à l'implantation de l'application (Tâches, Sémaphores),
- Tracer les graphes d'activité pour les différentes situations envisageables à savoir modifiez la priorité relative des tâches (*Horloge*, *IncTab* et *DecTab*) et en changeant l'implantation de *H* (mode priorité et FIFO pour la synchronisation),
- Préparer le programme C permettant l'implantation de l'application en utilisant VxWorks. Pour cela, vous pouvez vous inspirer de l'exemple fourni dans la documentation de Workbench,

Cette préparation doit être consignée dans un document (un document par binôme) qui sera relevé lors du début du TP.

3. IMPLANTATION

Afin de faciliter l'observation, il peut être intéressant, selon les situations, d'ajuster la période de l'horloge (paramètre de *taskDelay*) et la taille de Table.

Pour les TP en B013, vous pouvez, par exemple, utiliser *taskDelay(2)* pour le code de la tâche Horloge et une taille de 100000 éléments (int) pour la variable Partagée TABLE afin d'obtenir des résultats représentatifs (la taille de la table dépend, entre autre, de l'architecture interne du processeur qui exécute l'application et de la mémoire où est stockée la variable TABLE).

3.1. Mise en œuvre de la chaîne de développement et de simulation

En guise de premier travail, il convient de suivre le flot indiqué par la suite avec l'application exemple pour développer, compiler et simuler une application. Il est primordial pour vous de faire, en les comprenant bien, toutes les étapes requises pour développer et exécuter le code permettant d'implanter l'application. Ce flot sera repris par la suite. Remarquez bien que l'application exemple est proche (en partie) du code de l'application à développer pour le TP1.

3.2. Première étape : implantation de la synchronisation

Dans un premier temps, implanter seulement les fonctions *Horloge*, *DecTab* et *IncTab* ainsi que les relations *H* et *Table* (sans exclusion mutuelle). Dans cette première étape le plus simple est de limiter la taille de table à 5 éléments par exemple. Vérifier le comportement de votre solution à l'aide de System Viewer. Il s'agit donc de tester les cas où la fonction Horloge est la plus prioritaire puis les situations où les fonctions DecTab et IncTab sont plus prioritaires l'une que l'autre (dans les 2 sens) et de même priorité. Pour chacun des cas, tester

les 2 situations pour l'implantation de H, à savoir un seul sémaphore utilisant les modes FiFo et Priorité.

Comme le montre les transparents du cours, l'activation de n tâche par une synchronisation nécessite l'implantation d'une synchronisation par n sémaphores. Implanter cette solution dans le cas du TP et analysez les différentes situations présentées précédemment.

3.3. Deuxième étape : implantation de l'exclusion mutuelle

Augmenter la taille de la relation *Table* et ajouter la fonction *Contrôleur* dans un premier temps sans exclusion mutuelle pour *Table*. L'exécution du code montre alors les défauts d'une solution sans exclusion mutuelle, vous devez être en mesure de les expliquer (par exemple en utilisant un print et SystemViewer).

Mettre en œuvre une protection en exclusion mutuelle de la variable *Table* dans un premier temps en utilisant un sémaphore d'exclusion mutuelle, puis éventuellement par changement de priorité.

Dans le cas d'une implantation de l'exclusion mutuelle par sémaphore, expliquer les différentes prises de sémaphores qui peuvent apparaître durant l'activité de la tâche *Contrôleur*.

Afin de faciliter l'observation, il peut être intéressant, selon les situations, d'ajuster la période de l'horloge (paramètre de *taskDelay*) et la taille de *Table*.

Manip 2 : Attente passive

Cette manipulation a pour objectif :

- de montrer la différence entre attente active et attente passive,
- de mettre en évidence l'importance de l'ordre d'exécution des instructions,
- d'analyser la nécessité d'une exclusion mutuelle.

1. SITUATION

Il s'agit d'implanter une primitive, *Att_Passive(i)*, permettant de mettre en sommeil (attente passive) une tâche pendant une durée déterminée, définie en nombre de périodes Horloge. Cette primitive est à lier avec les différents mécanismes de délai, et de chien de garde mis à disposition par les exécutifs.

La structure fonctionnelle pour la situation où une seule tâche peut utiliser la primitive *Att_Passive* est la suivante :

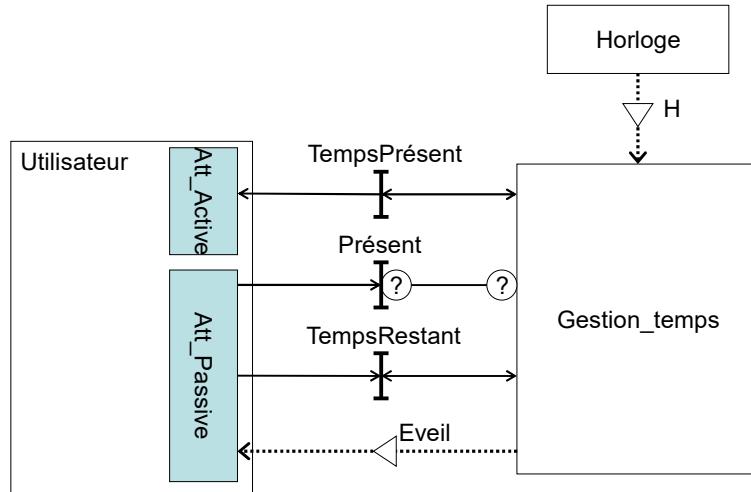


Figure 1 - Structure fonctionnelle pour le mécanisme *Att_Passive*.

Avec :

Gestion_temps : fonction temporaire activée par l'horloge (*taskDelay*) et chargée d'éveiller *Utilisateur* lorsque la durée de sommeil est écoulée,

Présent : variable indiquant qu'une demande est en cours,

TempsRestant : variable indiquant le nombre de tops Horloge restant avant l'éveil de l'utilisateur.

TempsPrésent : variable indiquant le nombre de tops Horloge écoulés depuis le départ de l'application. Ce temps n'est qu'un compteur initialisé et incrémenté par *Gestion_temps*. Il sera utilisé pour réaliser l'**attente active**.

Un exemple d'algorithme simple pour l'*Utilisateur* pourrait être :

```

action Utilisateur
    avec ...
    var date_sommeil, date_eveil : time;
begin
    cycle
    begin
        Att_Passive(3);
        Att_Active(5); // important d'avoir une durée différente de l'attente passive
    end;
    end_cycle;
end_utilisateur;

```

où *Att_Passive* a été définie en TD et *Att_Active* est la procédure C suivante :

```

void Att_Active(int DureeAttente)
{
    int DateEveil;

    if (DureeAttente > 0)
    {
        DateEveil = TempsPrésent + DureeAttente;
        while(TempsPrésent < DateEveil);
    }
};

```

2. TRAVAIL A EFFECTUER

2.1. En préparation du TP

- Indiquer la nature des accès à la variable *Présent* par *Gestion_Temps* (lecture, écriture ou lecture/écriture),
- Spécifier sous forme de réseau de Pétri le comportement des fonctions *Att_Passive* et *Gestion_Temps*,
- Définir les éléments VxWorks requis par l'application (Sémaphores, tâches, ...). Si la solution retenue est convenable, il n'est pas utile dans ce TP et pour la solution à 1 utilisateur de protéger les variables partagées par des exclusions mutuelles,
- Ecrire le programme C utilisant VxWorks pour réaliser l'application (solution à 1 utilisateur),
- Préparer le passage à plusieurs utilisateurs en définissant les éléments à dupliquer.

2.2. Durant le TP

a. Préciser et implanter la solution à 1 utilisateur

- Créer un nouveau projet dans l'espace de travail et y implanter le programme C utilisant VxWorks pour réaliser l'application.
- -Utiliser System Viewer pour montrer la principale différence de comportement entre les procédures *Att_Passive* et *Att_Active*.

b. Etendre la solution à un nombre n d'utilisateurs

Il s'agit de multiplier le nombre d'utilisateurs de la primitive *Att_Passive* en ne gardant qu'une seule fonction *Gestion_temps*. Chaque utilisateur (Tâche application) peut effectuer un ou plusieurs appels à *Att_Passive*, avec des paramètres différents.

- Déterminer les éléments de la structure fonctionnelle qui doivent être dupliqués. Pour l'implantation, il est conseillé de regrouper ces éléments dans un tableau.
- Définir la nouvelle solution en affectant dynamiquement les éléments libres du tableau aux utilisateurs.
- Existe-t-il de nouveaux problèmes d'exclusion mutuelle à résoudre?

Transfert de messages

Cette manipulation a pour objectif :

- d'utiliser le mécanisme de boîte aux lettres,
- d'estimer l'utilité de l'exclusion mutuelle.

1. SITUATION

On suppose la structure fonctionnelle suivante :

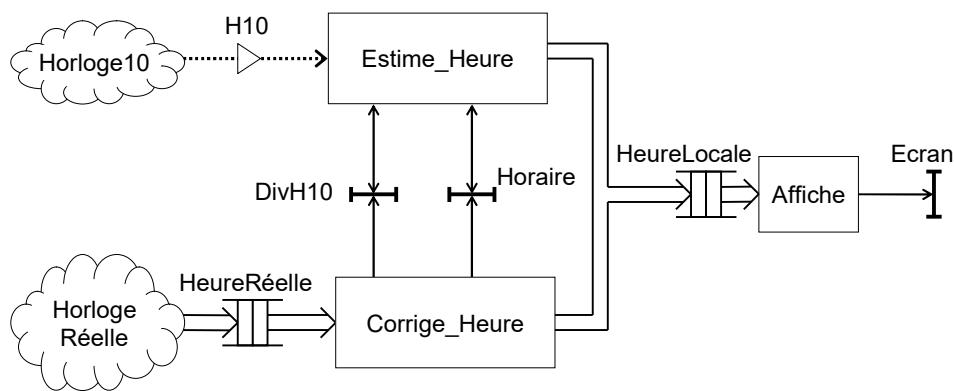


Figure 1 - Structure fonctionnelle pour Manip3.

L'application consiste à afficher sur *Ecran*, l'heure (heure, minute, seconde). L'heure peut être obtenue par estimation à partir d'une horloge à 10Hz (*H10*) et pour éviter les dérives, l'heure réelle est reçue par l'intermédiaire de *HeureRéelle* (heure, minute, seconde). Pour information, afin de limiter la longueur des observations, la base de temps de l'environnement est beaucoup plus rapide que la seconde.

1.1. Fonction Estime_Heure

Action activée sur *H_10*, événement périodique apparaissant 10 fois par seconde. Elle incrémente la variable *DivH10* si elle est inférieure à 9, sinon elle met la variable à 0 et la variable *Horaire* est incrémentée d'une seconde. Un message est transmis dans *HeureLocale*. Le message indique que la valeur transmise est une valeur estimée et contient la nouvelle valeur de *Horaire*.

1.2. Fonction Corrige_Heure

Action activée sur *HeureRéelle*. Elle copie la valeur reçue dans *Horaire*, met la variable *DivH10* à 0 et transmet un message dans *HeureLocale*. Le message indique que la valeur transmise est une valeur réelle et contient la nouvelle valeur de *Horaire*.

1.3. Fonction Affiche

Cette fonction copie sur *Ecran* l'heure reçue par l'intermédiaire de *HeureLocale* en indiquant s'il s'agit de l'heure estimée ou de l'heure réelle (c'est la seule tâche où vous pouvez faire un printf).

2. TRAVAIL A EFFECTUER

2.1. En préparation du TP

- -Spécifier sous forme de réseau de Pétri le comportement des fonctions *Estime_Heure* et *Corrige_Heure*,
- Définir les éléments VxWorks requis par l’application (Sémaphores, tâches, …),
- Ecrire le programme C utilisant VxWorks pour réaliser l’application,

2.2. Durant le TP

- -Définir, sous forme d’un type, la structure C des messages transmis par *HeureLocale*,
- Créer un nouveau péojet dans l’espace de travail et y implanter le programme C utilisant VxWorks pour réaliser l’application,
- Valider le comportement de votre code pour l’application à l’aide de WindView et montrer le principe de l’asservissement de la boîte aux lettres en limitant par exemple la taille de *HeureLocale* à 3 et en insérant des taskDelay dans Affiche. (questions à se poser : que se passe-t-il si la file de message est vide et si elle est pleine ?
- Montrer la différence de comportement qui peut découler du type de sémaphore (booléen ou compteur) utilisé pour implanter *H_10*.

L’environnement de votre application est simulé par un programme contenu dans le fichier *TP3Environment.o* (Horloge10 et HorlogeReelle). Le test de votre solution doit donc se faire en utilisant ce code objet et par lancement de l’environnement par appel à la procédure (fin de la procédure *Start*)

```
void startEnvironment(SEM_ID H10, MSG_Q_ID HeureRéelle)
```

dont le prototype est défini dans le fichier *TP3Environment.h*. Ce fichier contient aussi la déclaration de *type_heure* le type de la variable *Horaire* et des messages transmis par *HeureRéelle*. Le fichier ne comporte pas la déclaration du sémaphore *H10* et de la file de messages *HeureRéelle*, il vous revient de faire ces déclarations et de transmettre les références à l’environnement par le biais de la procédure *startEnvironment*.

3. VARIANTE

Afin de mettre en oeuvre le mécanisme de chien de garde mis à disposition par VxWorks pour les sémaphores, modifier la fonction *Estime_Heure* de façon à détecter l’absence d’événement *H10* pendant plus d’une seconde soit 11 événements. Dans ce cas, un message de type "Absence Horloge" (à définir) sera transmis dans *HeureLocale*.

L’environnement de l’application peut simuler ce cas. Il faut l’initialiser par la procédure

```
void startEnvironmentPb(SEM_ID H10, MSG_Q_ID HeureRéelle)
```

au lieu de la procédure

```
void startEnvironment(SEM_ID H10, MSG_Q_ID HeureRéelle).
```

Utilisation simplifiée de l'environnement *Workbench* pour les Travaux Pratiques d'Exécutif Temps-Réel

Olivier Pasquier, Janvier 2017

Ce document présente l'utilisation de l'environnement *Workbench* dans le cadre des Travaux Pratiques de 4^{ème} année du département ETN. *Workbench* étant une plateforme très complète, il faut être conscient que cette présentation est volontairement limitée aux seules fonctionnalités requises par les Travaux Pratiques et est donc loin d'être exhaustive.

Workbench est une plateforme, développée par *WindRiver*, basée sur le standard de plateforme *Eclipse*. Elle permet la définition et la mise au point d'applications temps-réel en se basant sur une gestion de projet comme le proposent d'autres outils de développement de logiciel généraux (Visual Studio, Visual Café ou Net Bean par exemple).

Le développement d'applications temps-réel à l'aide de *Workbench* nécessite a priori deux types d'ordinateurs reliés par une liaison série, USB, ou un réseau Ethernet par exemple.

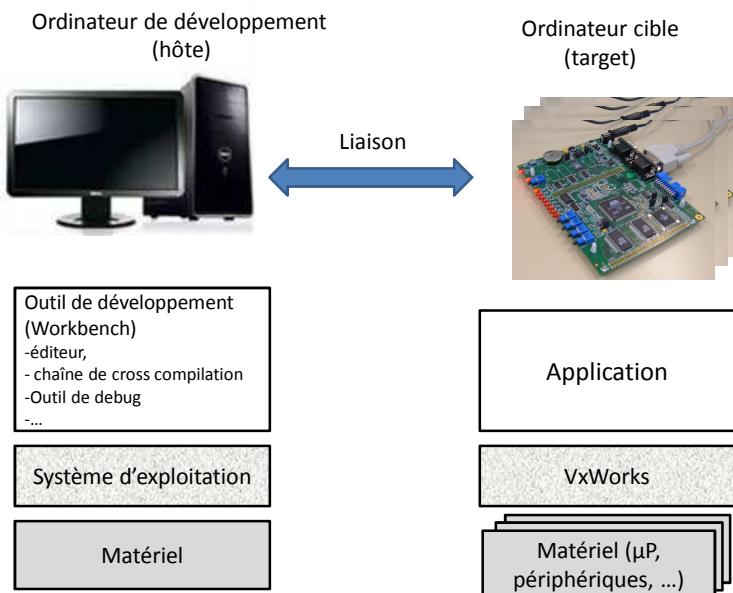


figure 1 Exemple de plateforme de développement.

La figure 1 présente un exemple d'organisation utilisant 2 ordinateurs, comme ce pourrait être le cas dans les travaux pratiques en ETN4 (cas du projet de microprocesseur par exemple).

- un ordinateur de développement aussi appelé ordinateur hôte (host dans la documentation) qui a un système d'exploitation (*Windows* ou *Linux* par exemple) sur lequel l'outil de développement *Workbench* peut être exécuté. Il s'agit d'un PC dans le cas du TP, mais dans un cas plus général, ce peut être aussi une station de travail (Sun, HP, IBM, ...). L'environnement *Workbench* est exécuté par cet ordinateur.
- un ou plusieurs ordinateurs cibles (target dans la documentation) sur lesquels

VxWorks est utilisé comme système d'exploitation. Il s'agit de l'ensemble d'ordinateurs qui va réellement exécuter l'application temps-réel développée. Cet ensemble d'ordinateurs peut être limité à une simple carte à microprocesseur, telle que la carte SAMD21 utilisée en projet de Systèmes à Microprocesseurs 16/32 bits, qu'un autre PC démarré avec l'OS *VxWorks* (et non pas *Windows*) ou encore être composé de plusieurs cartes reliées par un bus VME ou CompactPCI ou un réseau ethernet par exemple.

Afin d'éviter des configurations matérielles trop lourdes à mettre en œuvre, *Workbench* permet de simuler le comportement de la cible sur l'ordinateur hôte et donc de limiter le développement à l'utilisation d'un seul ordinateur (l'ordinateur hôte) sur lequel est exécuté à la fois le système de développement (*Workbench*) et le simulateur (*VxSim*) qui simule le comportement de la cible avec *VxWorks*. C'est cette solution qui a été retenue pour les Travaux Pratiques ETN4. Elle présente le défaut de ne pas permettre la connexion à l'environnement réel (moteur, ...) mais reste très intéressante pédagogiquement.

La suite de ce document présente succinctement les différentes étapes de la création, du développement et de l'analyse (debug) d'un projet dans *Workbench*. Pour cela, il faut dans un premier temps définir un espace de travail, puis définir dans cet espace de travail un projet. Il est à noter qu'un espace de travail peut contenir plusieurs projets. Par exemple il est possible de créer un espace de travail pour l'ensemble des TP et dans cet espace de travail créer plusieurs projets (TP1, TP2, TP3 par exemple). Une fois le projet créé, il convient de lui ajouter un ou plusieurs fichiers (.c ou .h), puis de compiler le projet. Le code exécutable ainsi créé doit être téléchargé sur la cible. L'exécution de ce code permet d'obtenir une observation du comportement de l'application à l'aide de l'outil *SystemViewer*. Commence alors le cycle de compréhension et de mise au point de l'application.

1 Démarrage de *Workbench*

Le démarrage de l'environnement de développement consiste à lancer *Workbench* à partir du menu *Démarrer* de Windows (groupe *Wind River* puis *Workbench3.3* et de nouveau *Workbench3.3*). L'outil va alors demander de définir l'espace de travail où vont se trouver vos projets comme le montre la figure 2 (le même espace de travail peut être utilisé pour les 3 Manips et pour stocker plusieurs variantes de chaque Manip). Il suffit alors de spécifier le répertoire dans lequel vous souhaitez travailler à l'aide du bouton « *Browse* » (①) en créant le répertoire si nécessaire.

Attention, dans la définition du chemin et du répertoire de travail il ne doit pas y avoir de caractères qui pourraient être considérés comme séparateur sous un autre OS que Windows (« espace », tabulation, ...). Il est conseillé de n'avoir que des caractères basiques (lettre, chiffre, -, _) et aucun caractère avec accent.



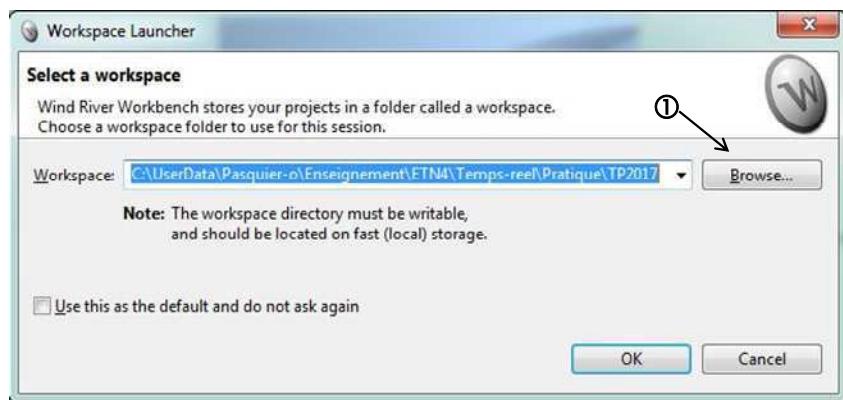


figure 2 Sélection de l'espace de travail.

L'outil s'ouvre ensuite, comme le montre la figure 3, dans la perspective Basic Device Development, dans le cas contraire, il faut activer celle-ci (1). Il convient alors de fermer la fenêtre de « bienvenue » (2), puis la fenêtre de présentation de Workbench (3). L'outil est alors prêt à être utilisé.

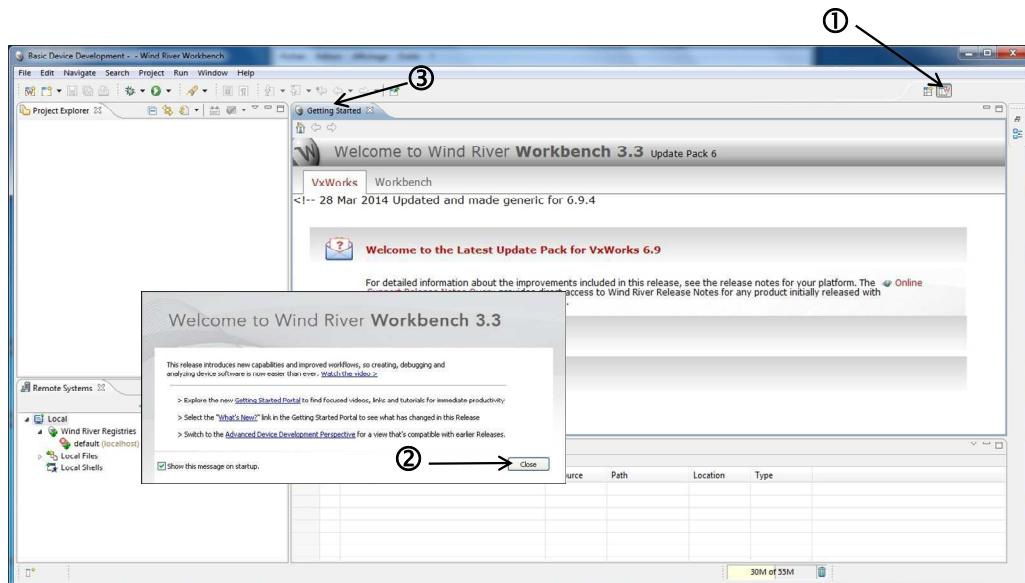


figure 3 Ouverture de Workbench.

Il est possible de définir plusieurs configurations (organisation des fenêtres) pour l'outil. Chaque configuration est appelée *Perspective*. La perspective par défaut (*Basic Device Development*) est montrée par la figure 4. Il est toujours possible de restituer cette configuration de la perspective en utilisant dans le menu principal *Window->Reset Perspective...*.

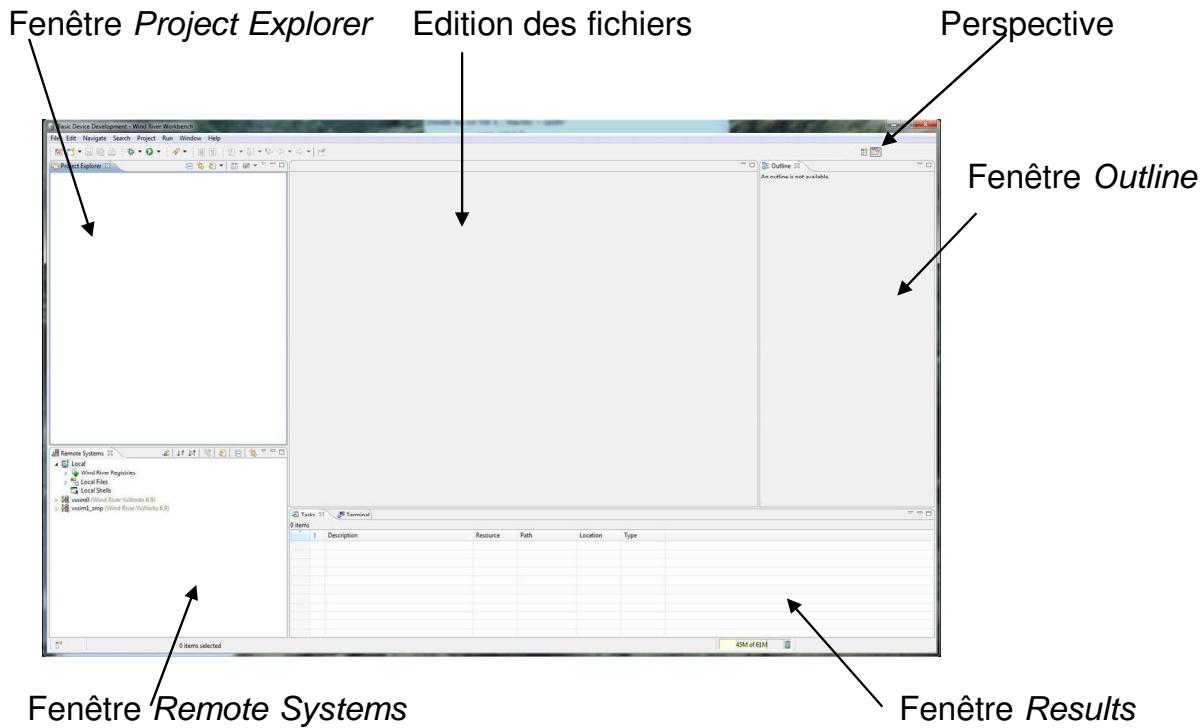


figure 4 Etat initial de l'environnement Workbench, perspective Basic Device Development.

Dans le cas de l'enseignement ETN4, nous utiliserons principalement les perspectives

Basic Device Development () et *System Viewer* (). La perspective *Basic Device Development* (voir figure 4) comporte une fenêtre *Project Explorer* dans laquelle est représentée l'arborescence des projets et des fichiers qui les composent, une fenêtre *Remote Systems* qui gère et représente l'état des cibles (simulateurs pour ces Travaux Pratiques), une partie pour l'édition des fichiers, une fenêtre *Outline* dans laquelle il est possible de voir les variables de l'application par exemple et une partie pour les résultats (*Results*).

2 Ouverture/Création d'un projet

L'utilisation de l'outil nécessite la définition d'un projet. Un projet est inclus dans un espace de travail. Chaque espace de travail peut lui-même comporter plusieurs projets c'est-à-dire, dans le cas des Travaux Pratiques ETN4, plusieurs manips ou plusieurs versions d'une même manip. Vous pouvez donc créer un seul et même espace de travail pour toutes les séances de pratique temps-réel.

2.1 Crédit d'un projet

La création d'un projet pour le développement d'applications sous VxWorks se fait classiquement par *File->New->Wind River Workbench Project...* comme le montre la partie droite de la figure 5.

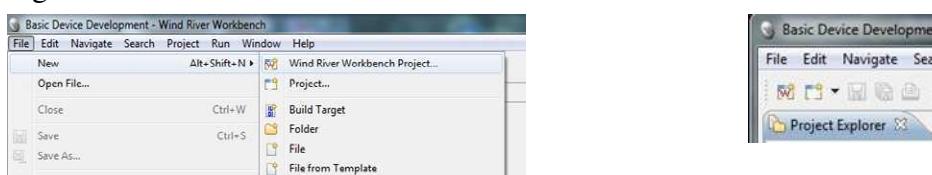


figure 5 Crédit d'un projet.

Il est aussi possible de créer un nouveau projet dans l'espace de travail en utilisant le bouton  dans la barre menu (partie droite de la figure 5).

L'outil demande alors le système d'exploitation cible. Il convient de sélectionner « Wind River VxWorks 6.9 » (① figure 6), puis de passer à l'étape suivante (② figure 6).

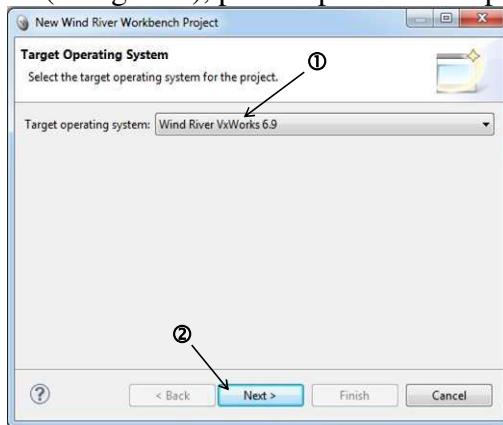


figure 6 Sélection du système d'exploitation cible.

L'outil demande alors le type de projet. Dans le cadre des Travaux Pratiques ETN4, le plus simple est de sélectionner « Downloadable Kernel Module » (① figure 7), puis de passer à l'étape suivante par *Next* (② figure 7).

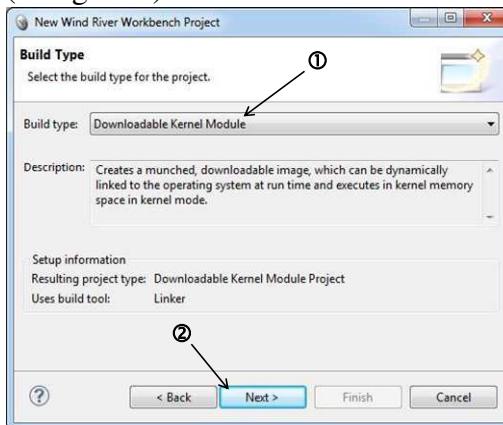


figure 7 Sélection du type de projet.

Il s'agit alors de définir le nom du projet (① figure 8), d'indiquer que le projet doit être ajouté à l'espace de travail courant (② figure 8) et enfin d'indiquer que la création du projet est terminée (③ figure 8).

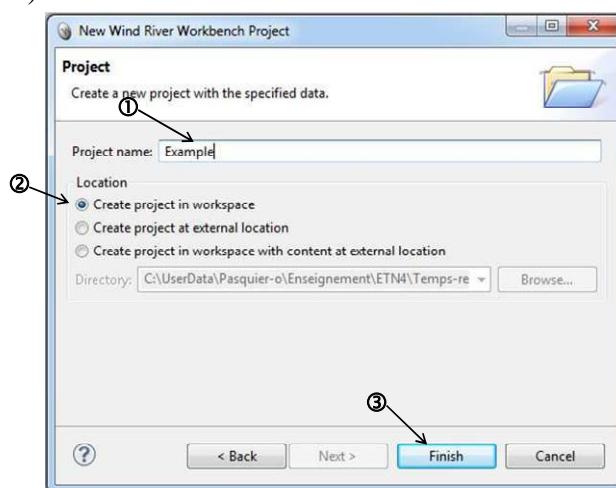


figure 8 Définition du nom du projet

Le projet est alors créé dans l'environnement *Workbench* (il apparaît dans le *Project*

Explorer de la perspective *Basic Device Development*) et sur le disque des fichiers, des répertoires sont créés ou modifiés pour prendre en compte ce nouveau projet. **Il va de soi qu'il est totalement déconseillé de faire des modifications directement sur le disque.**

2.2 Ouverture/fermeture d'un projet

Dans le cas où il y a plusieurs projets dans un même espace de travail, il est possible d'ouvrir ou de fermer les différents projets en sélectionnant le projet dans le *Project Navigator*, puis en cliquant bouton de droite puis *Open/Close Project*.

3 Insertion d'un fichier dans un projet

Lorsqu'un projet a été construit, il faut spécifier le ou les fichiers qui comportent le code de l'application. Cela peut se faire par sélection du projet dans la fenêtre *Project Explorer*, puis création d'un nouveau fichier par le menu *File* (① figure 9) puis *New* (② figure 9) puis *File* (③ figure 9). L'outil demande alors le nom du fichier, le crée, l'ouvre et l'ajoute au projet.

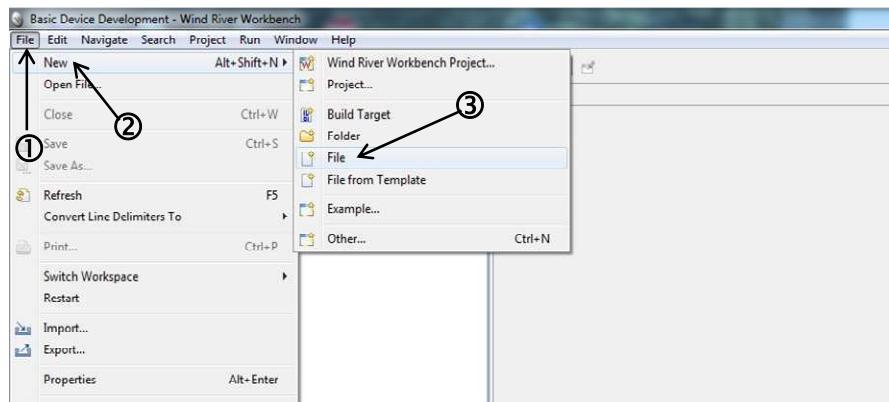


figure 9 Création d'un fichier source.

Il faut alors spécifier les caractéristiques du fichier qui doit être créé. Pour cela une nouvelle fenêtre est ouverte comme le montre la figure 10. Il faut spécifier le nom du fichier (①) et indiquer le répertoire projet concerné (②) par le stockage du fichier.

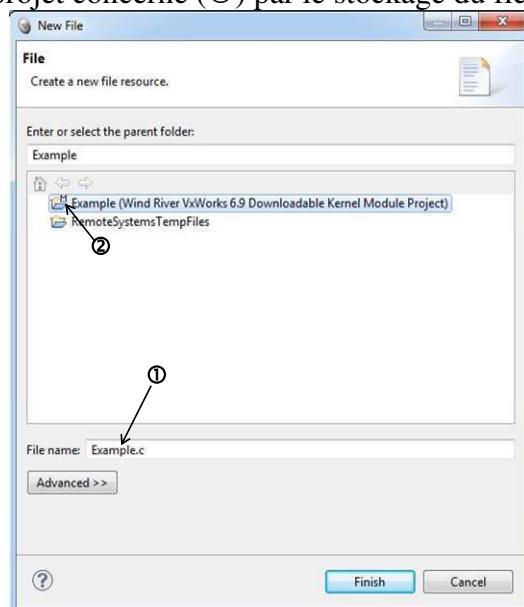


figure 10 Définition des caractéristiques du fichier.

Une fois le fichier créé, le fichier est intégré dans le projet et est ouvert comme le montre la figure 11.

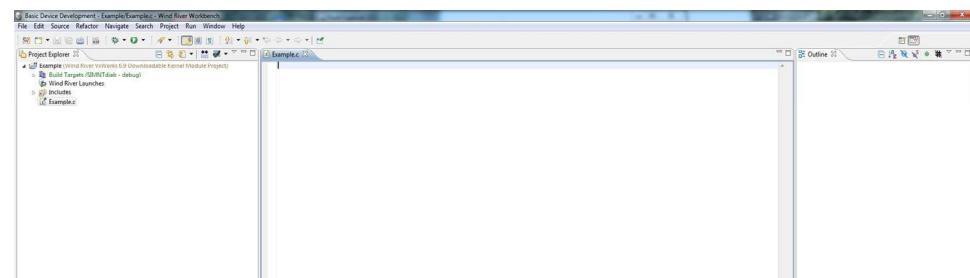


figure 11 Exemple de création de fichier dans un projet.

Il est aussi possible d'inclure un fichier déjà existant. Dans ce cas, il faut tout d'abord sélectionner le projet dans le *Project Explorer*. Ensuite, il faut soit faire *File* puis *Import* à partir du menu principal de l'outil (partie gauche de figure 12, ① puis ②), soit cliquer bouton de droite sur le projet dans la fenêtre *Project Explorer* puis *Import* (partie droite de figure 12, ③ puis ④).

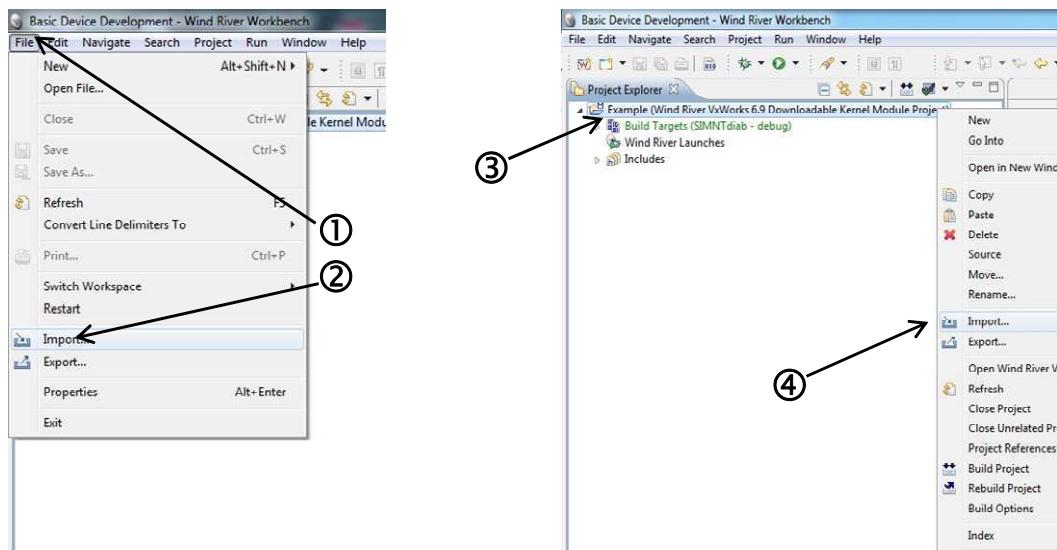


figure 12 Import d'un fichier dans un projet (début).

S'ouvre alors la fenêtre présentée à gauche de la figure 13 dans laquelle il faut étendre la partie « *Général* » (①) puis sélectionner *File System* (②) et passer à la fenêtre suivante (③). L'outil ouvre alors la fenêtre présentée à droite de la figure 13 et dans laquelle il faut dans un premier temps sélectionner le répertoire dans lequel se trouve le fichier à importer (④), puis le ou les fichiers (⑤) à importer et valider (⑥). **Il est à noter que la manipulation présentée, avec les options sélectionnées fait une copie du ou des fichiers importés dans le répertoire du projet.**

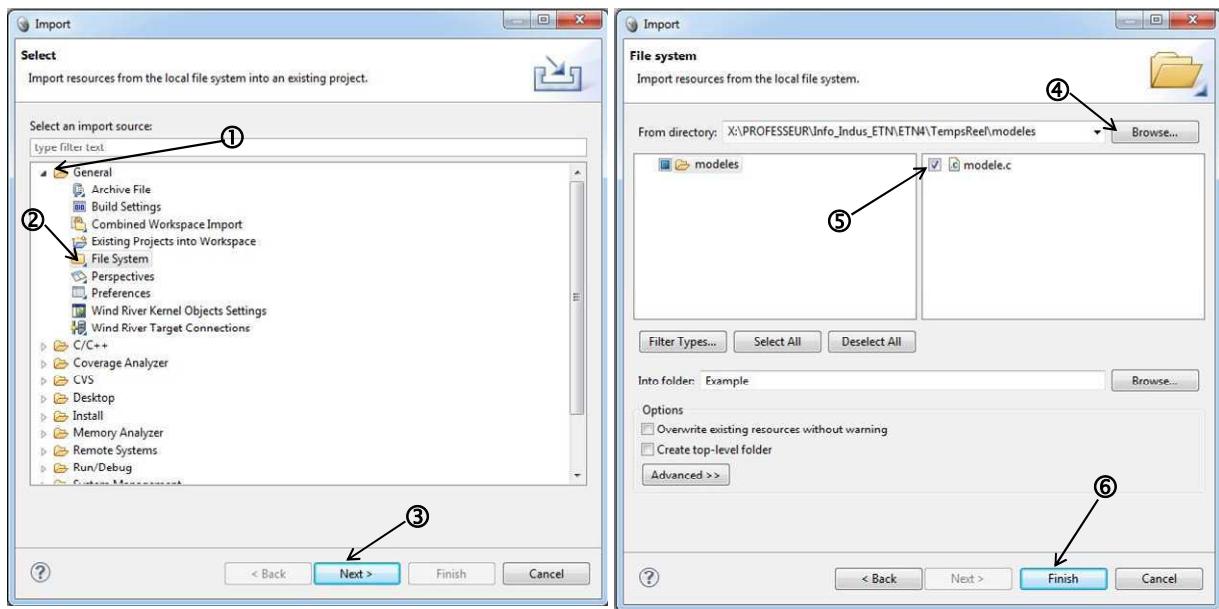


figure 13 Import d'un fichier dans un projet (fin).

A titre d'exemple, il est possible d'importer le fichier *modele.c* qui se trouve sur le volume *//irsmbetu/temp*, dans le répertoire *PROFESSEUR/Info_Indus_ETN/ETN4/Temps_Reel/modeles*. Ce fichier contient un exemple opérationnel d'application simple.

Contrairement à certains outils, le compilateur n'exploite pas l'extension du fichier (.c ou .cpp) pour reconnaître la syntaxe C ou C++ dans les commentaires. La syntaxe // qui est spécifique à C++ pour définir les commentaires est ici reconnue par le compilateur C.

Après l'importation ou la création du fichier, celui-ci apparaît dans la fenêtre « *Project Navigator* » de l'outil. Il est possible d'ouvrir le fichier en double cliquant sur son nom. La figure 14 montre un exemple d'ouverture du fichier *modele.c* qui a été importé précédemment.

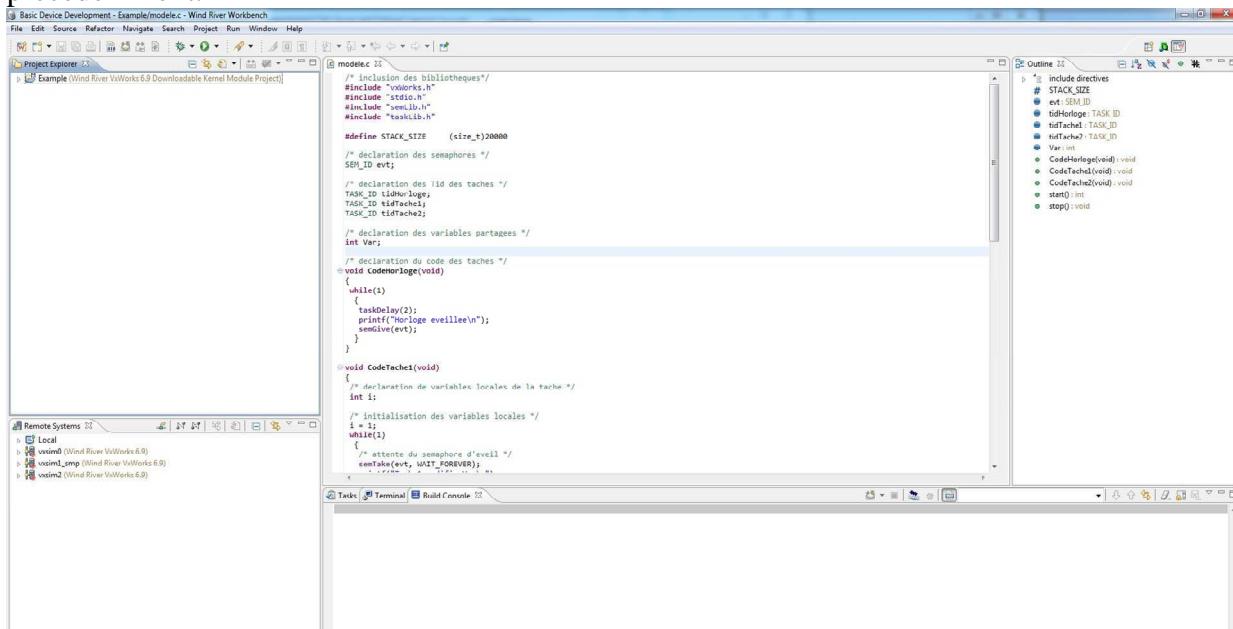


figure 14 Exemple d'édition d'un fichier.

La partie droite de l'outil présente une synthèse des éléments présents dans le fichier.

4 Compilation du projet

La compilation du projet est classique. Il faut, dans un premier temps, sélectionner le projet dans le *Project Navigator* (① figure 15). La compilation peut être effectuée

- soit à partir du menu principal *Project* puis *Build Project* (② puis ③ figure 15),
- soit en cliquant bouton de droite dans le *Project Navigator* puis *Build Project* (④ puis ⑤ figure 15),
- soit en utilisant le bouton  dans la barre du *Project Navigator* (⑥ figure 15).

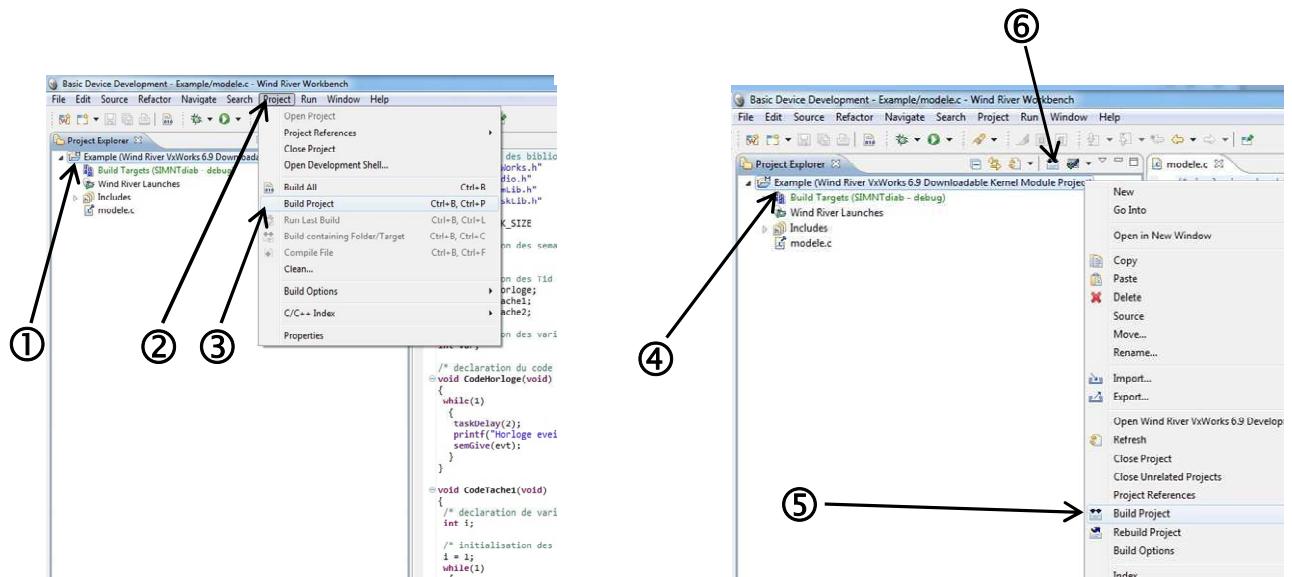


figure 15 Compilation du projet.

Lors du premier *Build* d'un projet, il est nécessaire de définir les contraintes à satisfaire pour obtenir le code final. Pour cela, *Workbench* ouvre un ensemble de fenêtres pour la création des dépendances du projet (fichiers à compiler et à inclure) et le choix de la cible (cf figure 16). Dans le cas des TPs ETN4, la dépendance est très simple et il suffit de cliquer *Generate Includes* (①). S'ouvre alors une première fenêtre dans laquelle il faut laisser les options par défaut, puis passer à la seconde fenêtre (*Next* ②) dans laquelle il convient d'avoir le même comportement pour atteindre la troisième fenêtre (*Next* ③). Il faut alors désélectionner toutes les cibles (④), sélectionner la cible SIMNT diag (⑤) puis lancer la compilation (*Finish* ⑥).

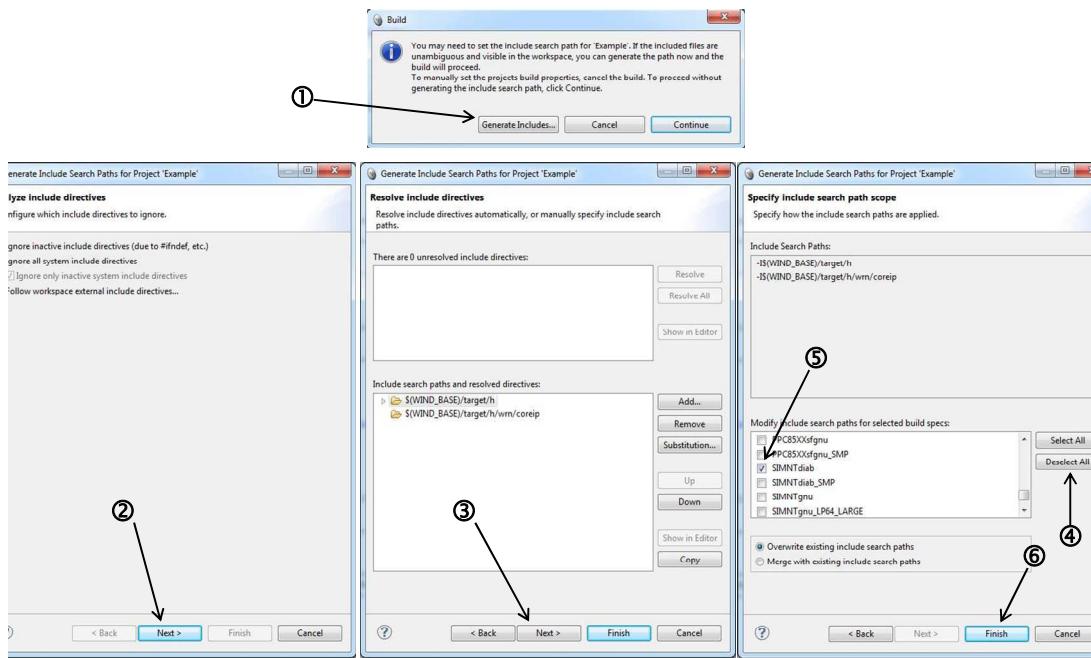


figure 16 Fenêtre pour la définition des dépendances.

La figure 17 présente un exemple de résultat de compilation. Les commentaires de la compilation sont affichés dans la fenêtre résultat dans l'onglet *Build Console* (1).

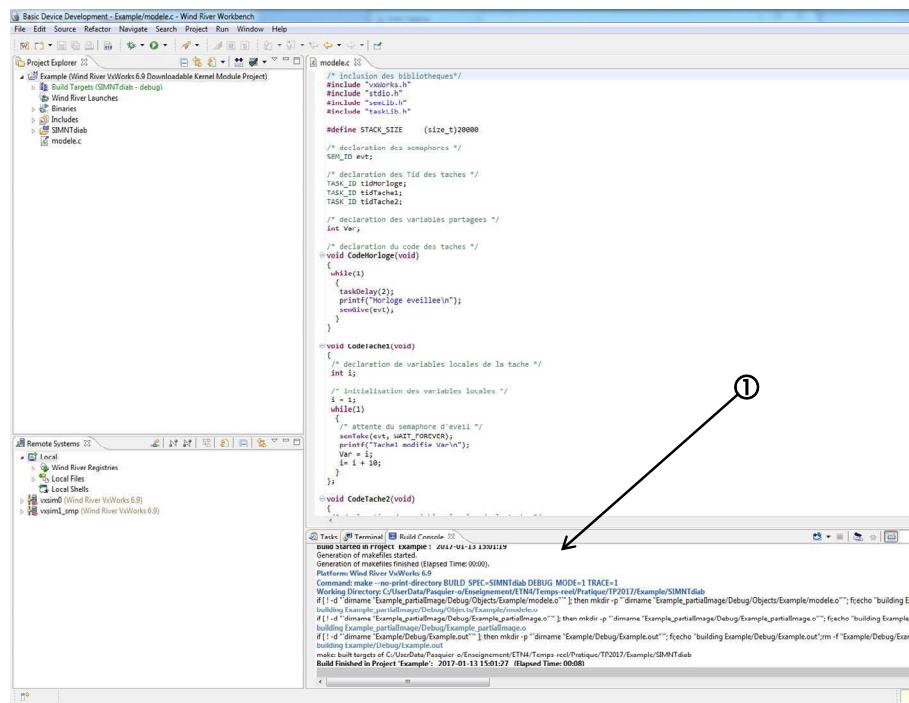


figure 17 Exemple de résultat de compilation.

5 Démarrage du simulateur

Dans le cas des Travaux Pratiques, nous n'utiliserons pas une cible réelle. Celle-ci sera simulée. Par défaut, Wind River fournit un simulateur appelé *VxSim*. Il est aussi possible de définir son propre simulateur et de l'utiliser comme nous le verrons plus tard (voir 10 Démarrage d'un simulateur spécifique)

Par défaut, *Workbench* a prédéfini deux connections pour des simulateurs dans la

fenêtre *Remote Systems* : *vxsim0* et *vxsim1_smp*. Le second, *vxsim1_smp*, vise plutôt des cibles multi-processeurs (smp pour Symmetric MultiProcessor). Il ne sera pas utilisé dans le cadre des Travaux Pratiques ETN4.

Pour des raisons pédagogiques, il est plus intéressant d'avoir une fenêtre dédiée au dialogue avec la cible. Pour cela, il faut modifier les options par défaut du projet. Il convient donc d'aller dans le menu principal de Workbench *Window* (① figure 18) puis *Preferences* (② figure 18).

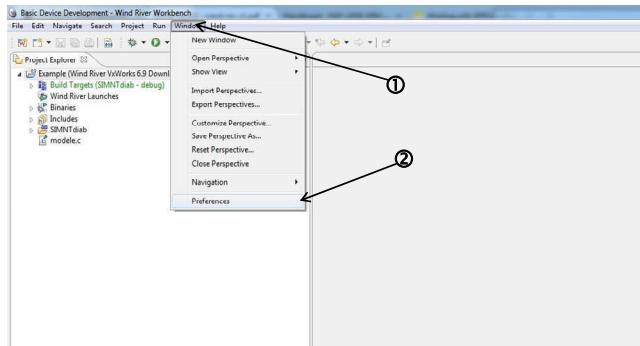


figure 18 Ouverture des options du projet.

Une nouvelle fenêtre s'ouvre (voir figure 19). Il s'agit pour nous d'indiquer qu'une console spécifique doit être associée au simulateur. Pour cela, il convient, sous *Wind River* (① figure 19), *Target Management* (② figure 19) de sélectionner *Target Console* (③ figure 19) et décocher *VxWorks Simulator Connections* (④ figure 19). La fenêtre de configuration peut ensuite être quittée (⑤ figure 19).

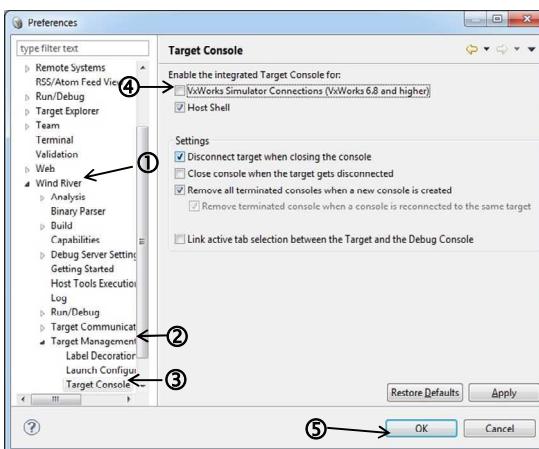


figure 19 Paramètre Target Console pour le simulateur.

Il est maintenant possible d'activer le simulateur *vxsim0*. Pour cela, il suffit de le sélectionner dans la fenêtre *Remote Systems* (① figure 20), puis de le connecter (② figure 20). Il est aussi possible de démarrer le simulateur en cliquant dessus avec le bouton de droite dans la fenêtre *Remote Systems* (③ figure 20), puis « connect *vsim0* » (④ figure 20)

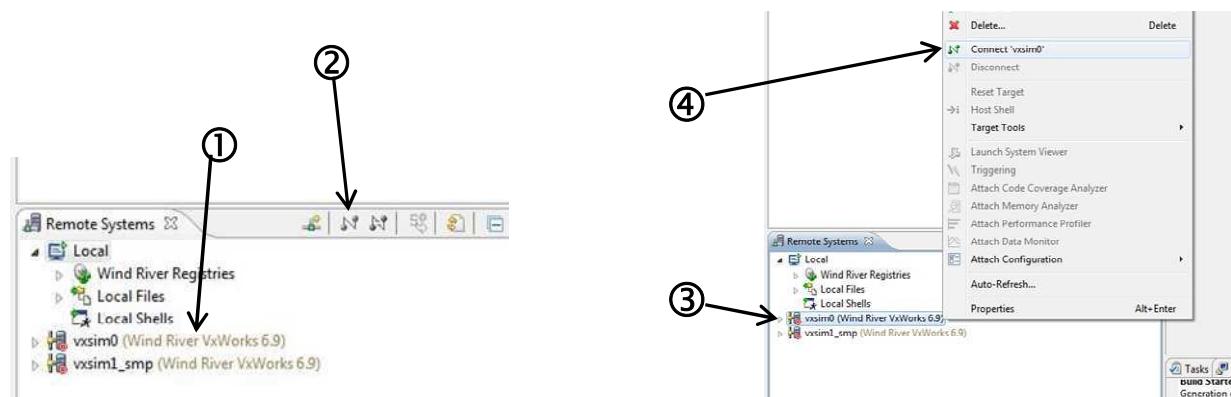


figure 20 Démarrage de vxsim0.

Dans les deux cas, une fenêtre windows (console) d'interface avec le simulateur s'ouvre (figure 21). Cette fenêtre est la seule fenêtre de la cible et sera utilisée pour tout dialogue avec celle-ci. Les résultats des opérations d'impression (*printf*) de votre application seront affichés dans cette fenêtre et c'est dans celle-ci qu'il est possible de faire des entrées clavier destinées à l'application (*scanf*, *getchar*, ...). Cette fenêtre est enfin utilisée pour donner des ordres à la cible (démarrage et arrêt de l'application par exemple).

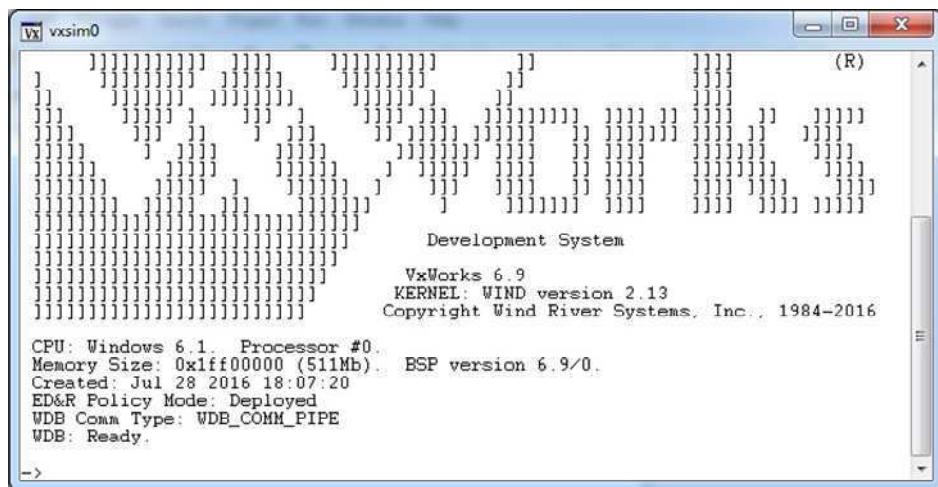


figure 21 Fenêtre interface du simulateur.

Dans la partie *Remote Systems* de l'outil *Workbench*, il est possible de valider les modules qui sont chargés sur la cible comme le montre la figure 22. Au démarrage, seuls les modules liés au noyau sont chargés (*Kernel Tasks*, *Real Time Processes*, *vxWorks*). Après un téléchargement de votre application, cette zone fera aussi apparaître le module téléchargé comme il sera montré plus tard dans le document.



figure 22 Etat de la cible.

Il est possible de réinitialiser le simulateur en entrant Ctrl-x dans sa fenêtre interface.

6 Démarrage et configuration de System Viewer

Pour mettre au point les applications temps-réel, il est possible d'utiliser un debugger comme pour la mise au point des applications classiques. Un tel outil est intégré dans *Workbench*. Pour des raisons pédagogiques, il ne sera pas utilisé dans le cadre de ces Travaux Pratiques ETN4. Nous utiliserons principalement *System Viewer* qui permet d'observer les propriétés temporelles et le comportement multi-tâches de l'application. Pour cela, il trace le diagramme temporel d'activité des tâches qui se trouvent chargées sur la cible (le simulateur dans notre cas). Son lancement se fait à partir de la fenêtre *Remote Systems* en sélectionnant la cible, puis en cliquant bouton de droite (① figure 23) puis *System Viewer Configuration* (② figure 23).

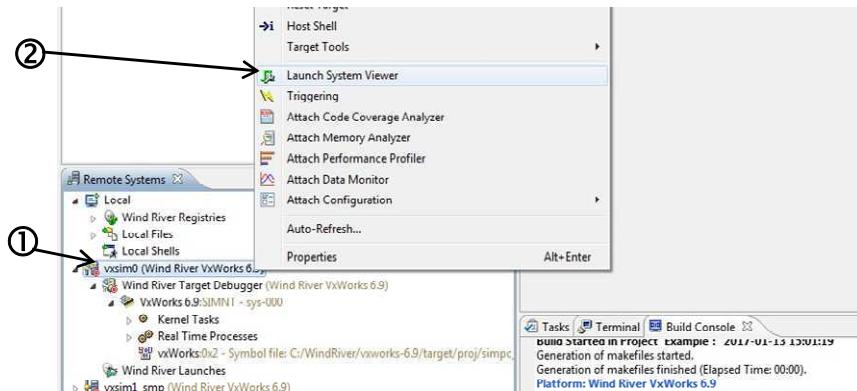


figure 23 Lancement de SystemViewer.

Apparaît alors la fenêtre présentée par la figure 24 dans laquelle il est important de noter les icônes (①) qui indiquent que la configuration de *System Viewer* n'est pas cohérente avec le simulateur.

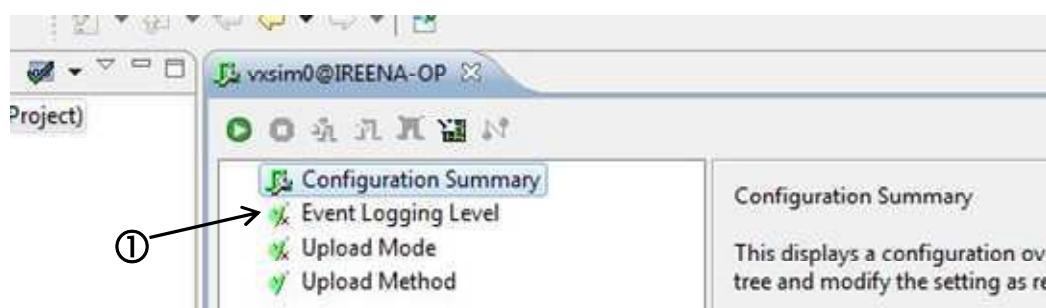


figure 24 Configuration de System Viewer.

La configuration de l'observation de l'application se fait à partir des menus *Event Logging Level* et *Upload Mode*. *Event Logging Model* (① figure 25) permet de spécifier ce que l'on souhaite observer sur la cible. Dans un cas général, il peut être difficile d'observer trop d'informations car la liaison entre la cible et l'hôte peut ne pas avoir un débit suffisant, ou sa gestion peut perturber le comportement temps-réel de l'application. Dans le cas du TP, il faut configurer le mode *Additional Instrumentation* (② figure 25). L'outil permet ensuite de sélectionner les éléments à observer. Dans le cadre des Travaux Pratiques ETN4, il faut au moins sélectionner *Task*, *Semaphore*, *Message Queue*, *Watchdog* et *ISR*.

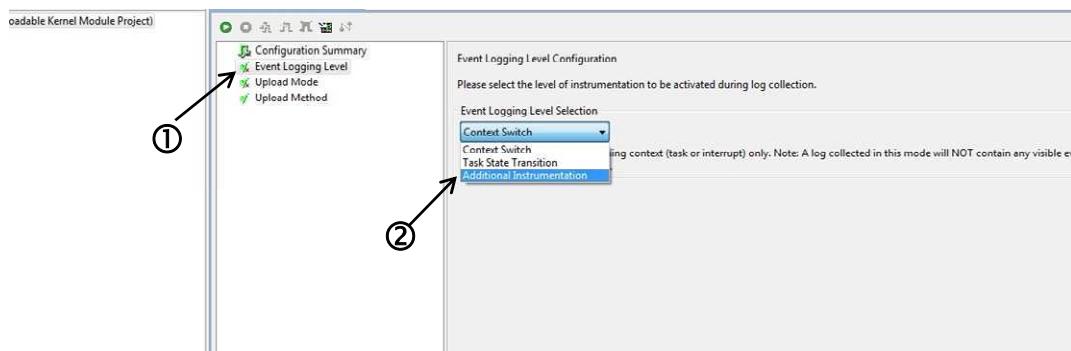


figure 25 Configuration des informations à observer.

Il faut aussi configurer le principe de récupération des informations (*UpLoad*) par le menu *Upload Mode* (① figure 26). La configuration concerne l'instant de récupération des informations ainsi que la taille du buffer qui doit être réservé sur la cible. Dans le cadre des Travaux Pratiques ETN4, il vous est conseillé d'utiliser la configuration présentée figure 26 : Deferred Upload (②), Buffer Size égale à 256koctet (③) (attention, cette taille influence la taille de l'image vxWorks qui serait chargée sur une cible réelle).

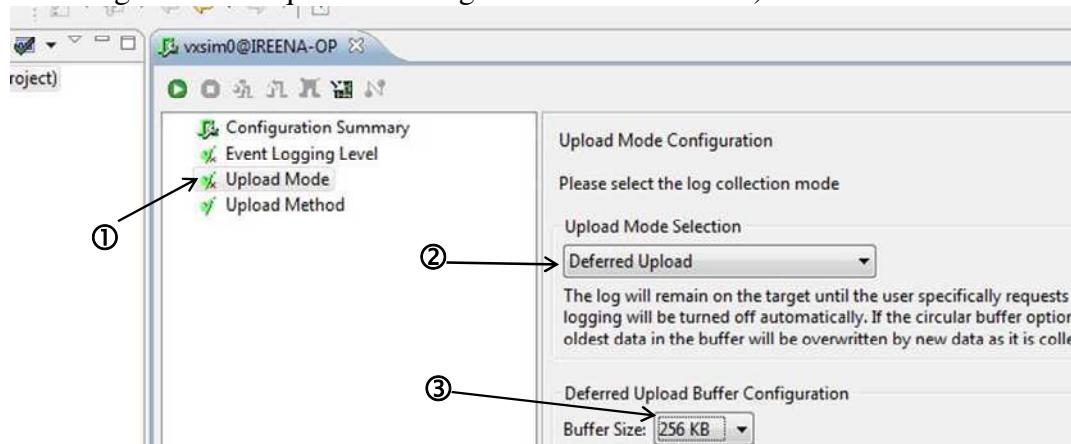


figure 26 Configuration de la récupération des événements.

Une fois la configuration terminée, il faut mettre à jour la configuration de *VxWorks* sur la cible en utilisant le bouton du menu de configuration de *Wind View* (① figure 27). L'outil doit alors remplacer les icônes par (②).

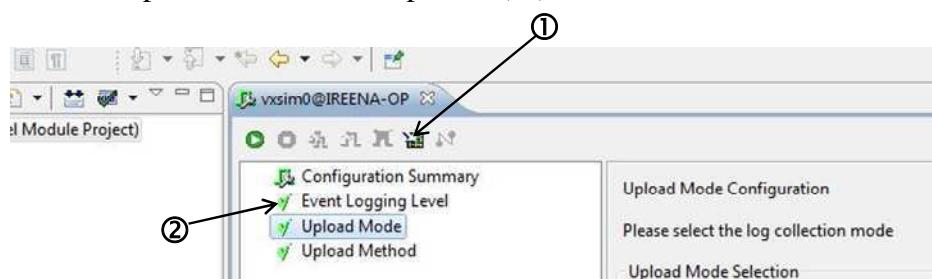


figure 27 Mise à jour de la configuration de System Viewer.

Une nouvelle observation sur la cible n'implique pas d'arrêter puis démarrer *System Viewer*. Par contre, en cas d'arrêt puis démarrage de la cible, ou en cas de reset de celle-ci, il est fortement recommandé d'arrêter puis redémarrer *System Viewer*.

7 Téléchargement de l'application.

7.1 Cas général

Les étapes précédentes ont permis de lancer tous les outils nécessaires au développement et à l'observation de l'application. Il s'agit maintenant de télécharger l'application sur la cible (le simulateur dans le cas des Travaux Pratiques ETN4). Pour cela, l'opération consiste d'abord à sélectionner la cible (① figure 28) dans la partie *Remote Systems*. Ensuite, sélectionner le projet à télécharger dans la partie *Project Navigator* (②), puis en cliquant sur le bouton de droite, lancer l'opération *Download* (③) puis *VxWorks Kernel Tasks* (④).

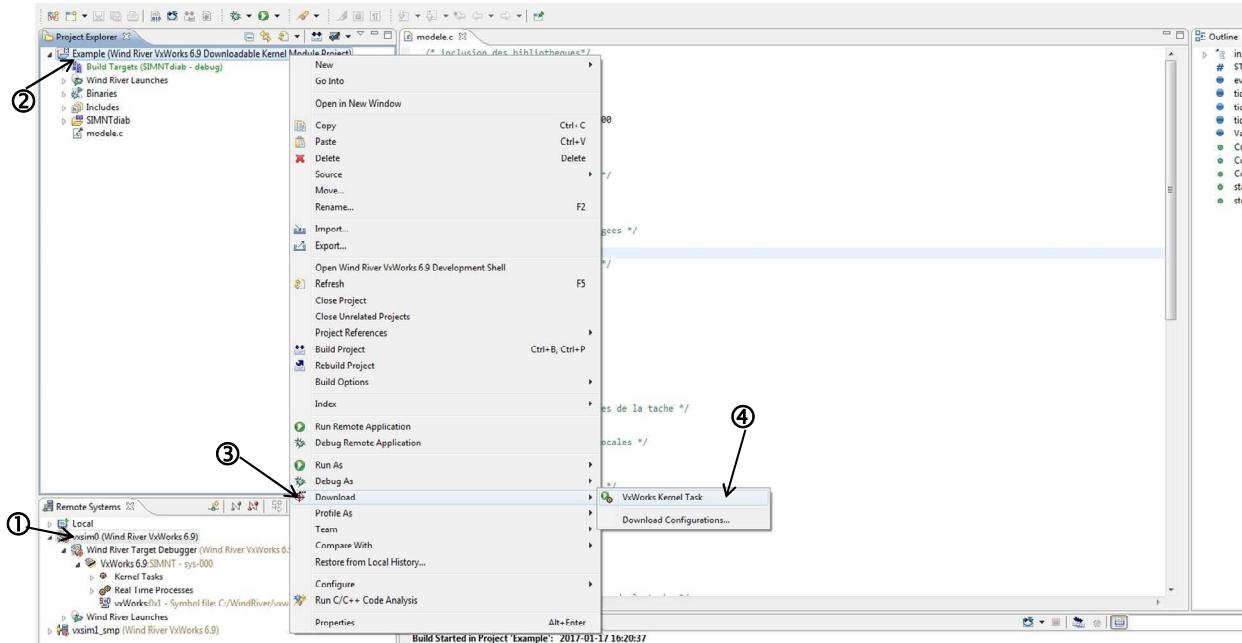


figure 28 Téléchargement d'une application.

Une nouvelle fenêtre s'ouvre alors (figure 29). Elle permet de définir une nouvelle configuration de téléchargement. Pour cela, il convient dans un premier temps de définir son nom (①), ici à titre d'exemple le nom du projet et le nom de la cible. L'onglet *Launch Context* (②) a pour objectif de définir la cible sur laquelle on souhaite télécharger du code et l'onglet *Downloads* (③) a pour objet de définir le code que l'on souhaite télécharger. Une fois les différents champs complétés, il faut enregistrer la configuration en cliquant sur *Apply* (④), puis effectuer le téléchargement en cliquant *Download* (⑤). Dans le cas où la configuration a déjà été enregistrée, évolution du programme par exemple, seule cette dernière opération est requise.

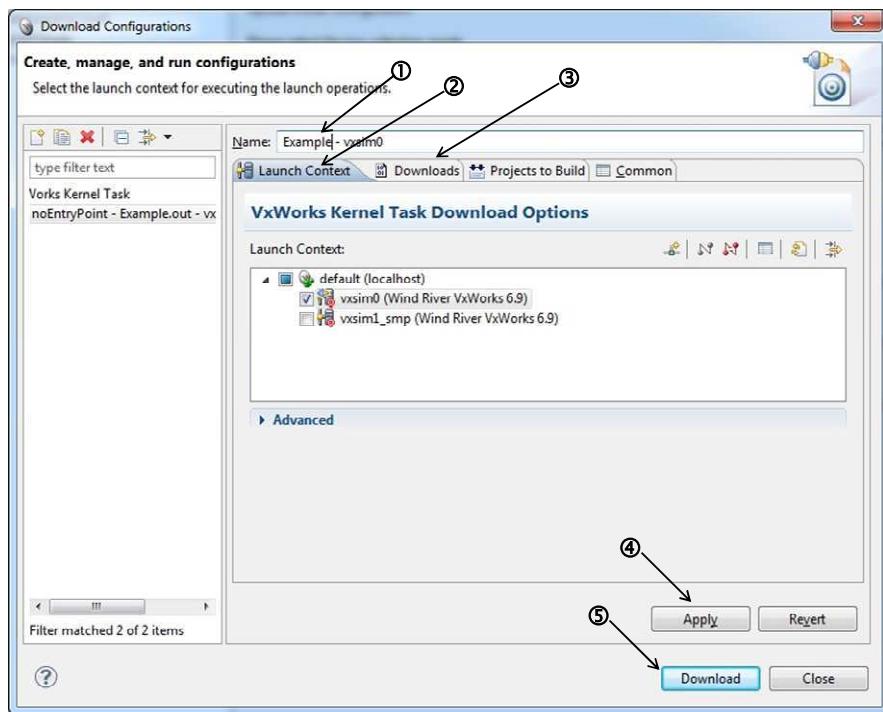


figure 29 Téléchargement d'un code sur le simulateur.

Il existe de nombreuses autres solutions, dont une qui consiste à définir une configuration de téléchargement (voir figure 29 *Download Configuration*), puis à utiliser cette configuration pour effectuer les téléchargements. Cette solution est souvent utilisée implicitement car les configurations de téléchargement sont automatiquement sauvegardées. Il est à noter que quelle que soit la démarche utilisée, le module chargé doit apparaître dans la partie *Remote Systems* comme le montre la figure 30.

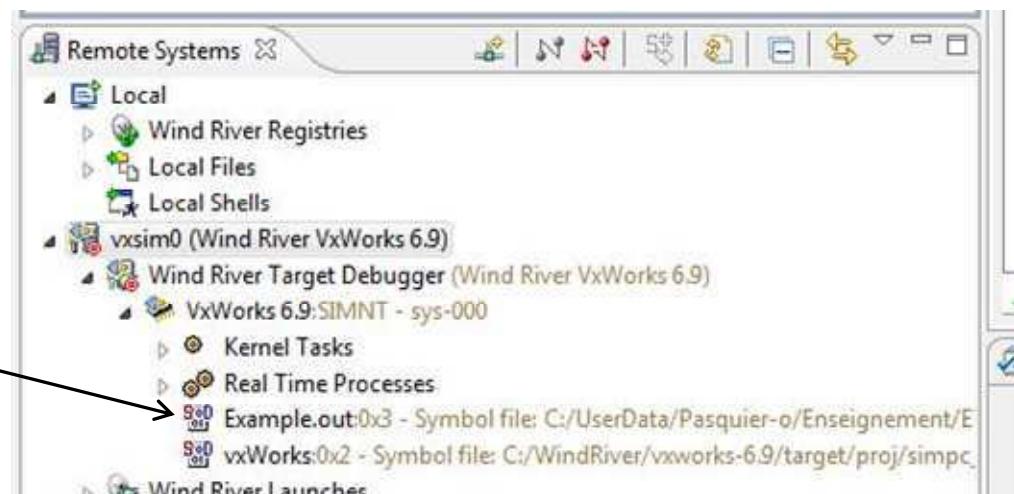


figure 30 Contenu de la cible après le téléchargement.

7.2 Cas particulier du TP3

De façon générale et plus particulièrement dans le cadre du TP3, il peut être nécessaire de charger un code objet particulier (TP3Environment.o par exemple) en respectant un ordre de chargement car l'édition de lien se fait au chargement des fichiers, ce qui implique que le code de toute procédure appelée par le code téléchargé doit déjà être disponible sur la cible. Pour cela, il convient dans un premier temps de définir une configuration de téléchargement

qui contienne tous les fichiers requis et définit leur ordre de chargement.

Il faut donc, comme le montre figure 31, dans la fenêtre *Remote Systems* ouvrir la procédure de chargement qui a été définie en cliquant sur les « + » en vis-à-vis de la cible sélectionnée, de « *Wind River Launches* » et de « *VxWorks Kernel Tasks* » (1). Cliquer bouton de droite sur la procédure de chargement définie (2), sélectionner « *Open As* » (3), puis « *Download Configuration...* » (4).

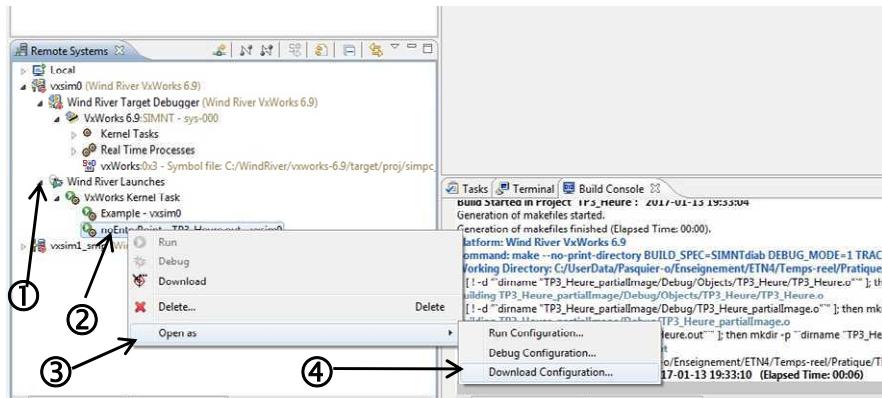


figure 31 Modification d'une configuration de téléchargement.

Une nouvelle fenêtre, telle que celle présentée par la figure 32, s'ouvre alors. Il faut tout d'abord sélectionner la configuration à modifier (1), puis éventuellement modifier son nom (2), sélectionner la partie « *Downloads* » (3), ce qui conduit à la figure 33 (partie gauche).

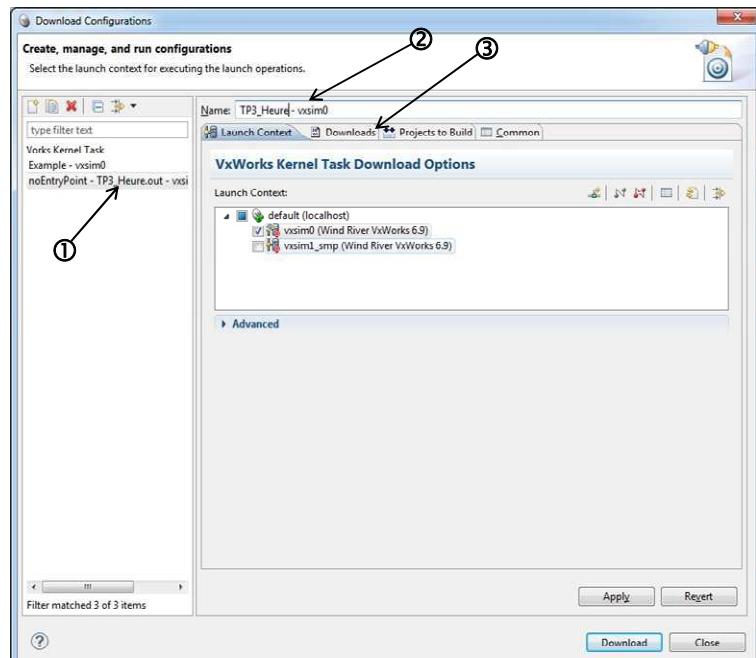


figure 32 Définition d'une nouvelle configuration.

Dans le cas du TP3, il s'agit d'indiquer les fichiers à télécharger, ainsi que spécifier le bon ordre de téléchargement car l'édition de liens se fait au moment du téléchargement, donc un mauvais ordre va conduire à des messages indiquant que des procédures sont absentes. Pour ajouter un fichier, cliquer « *Add* » (1). Une nouvelle fenêtre s'ouvre alors (présentée dans la partie droite de la figure 33) dans laquelle il est possible d'indiquer le fichier à ajouter (2), le nom complet (chemin inclus) apparaît dans le champ (3). Le choix doit ensuite être

validé (④). La fenêtre se ferme. Il faut alors respecter l'ordre de chargement des fichiers pour que l'édition de liens qui se fait au chargement de chaque fichier puisse s'effectuer sans problème. Pour cela, il faut sélectionner les fichiers (⑤) puis les déplacer en utilisant les boutons « Up » et « Down » (⑥). Avant d'effectuer un téléchargement, il convient d'enregistrer les modifications en cliquant *Apply* (⑦).

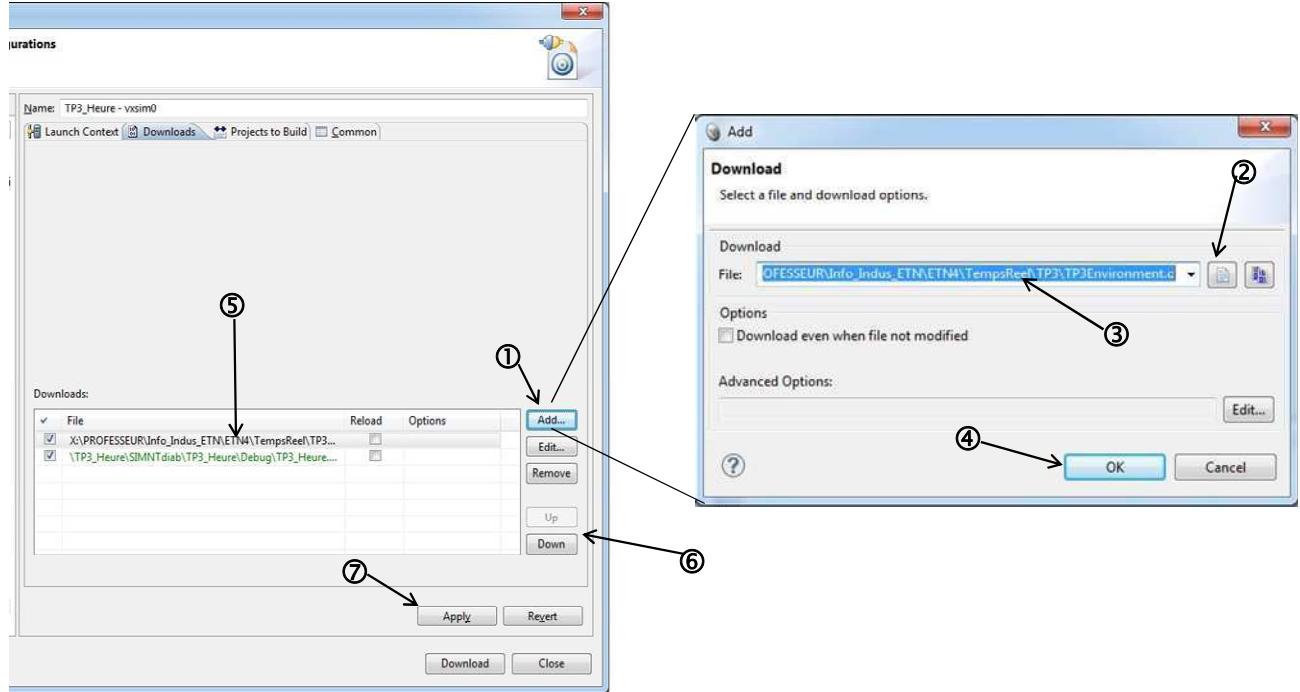


figure 33 Ajout d'un fichier à une configuration de téléchargement.

Pour le chargement de l'application, il suffira alors de double-cliquer sur la configuration ainsi définie dans la fenêtre *Remote Systems* comme le montre la figure 34.

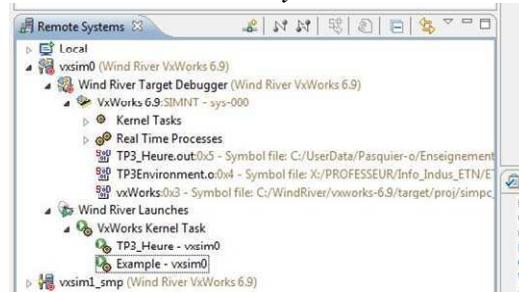


figure 34 Configuration de téléchargement.

8 Observation du comportement de l'application

Pour pouvoir observer le comportement de l'application, il faut, après avoir démarré *System Viewer* (cf paragraphe 6) et téléchargé l'application (cf paragraphe 7), lancer l'observation de l'application par *WindView* (bouton ① figure 35, (Attention il existe 2 boutons fortement semblables : **Start System Viewer Logging** et **Run ...** il faut ici bien sûr cliquer **Start System Viewer Logging**) puis lancer l'application (*start* dans la fenêtre d'interface de la cible ① figure 36).

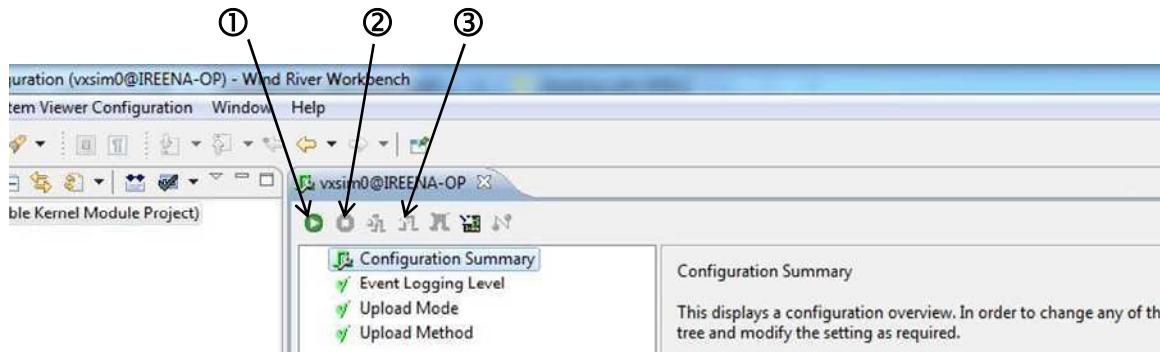


figure 35 Démarrage et arrêt de l'observation par SystemViewer.

Après quelques secondes, stopper l'application (stop dans la fenêtre d'interface de la cible ② figure 36), puis stopper l'observation par *System Viewer* (bouton ① ② figure 35). Enfin, il faut éventuellement récupérer les résultats de l'observation par le bouton *Upload* de *System Viewer* (③ figure 35).

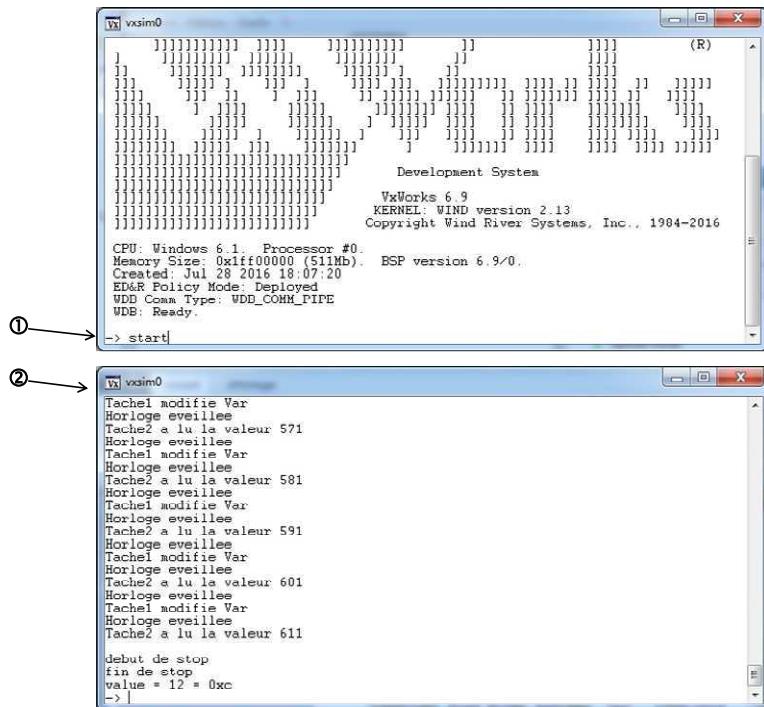


figure 36 Démarrage et arrêt de l'application.

Workbench récupère alors les informations concernant le comportement de l'application et les affiche comme le montre la figure 37.

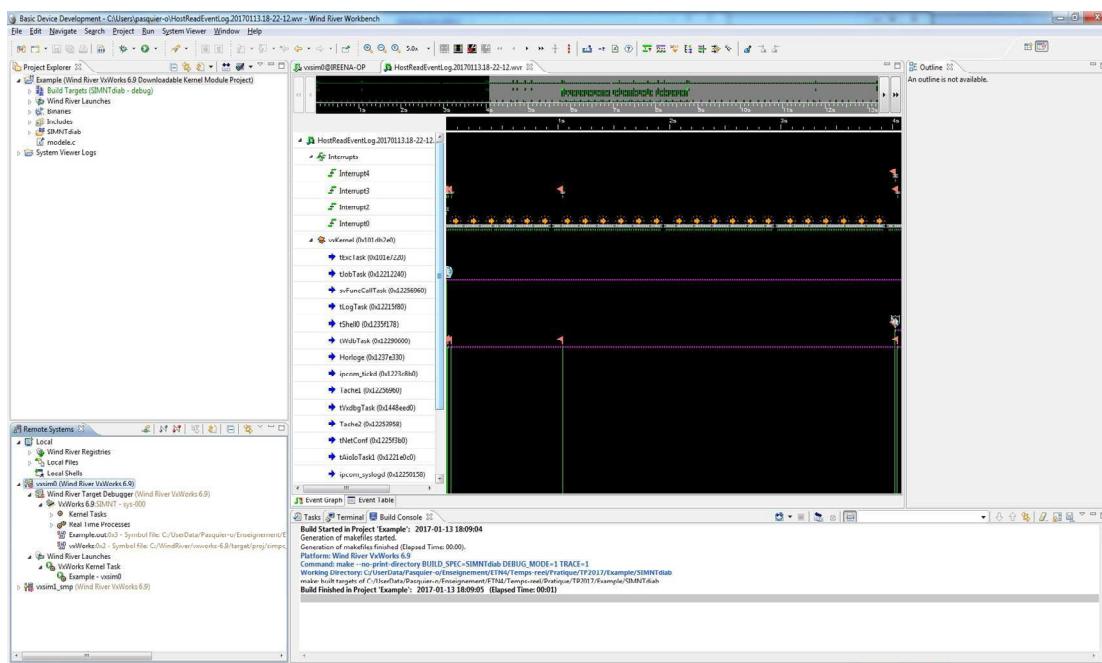


figure 37 Affichage des résultats par System Viewer.

Afin de faciliter le travail d'analyse de la trace, il est conseillé d'utiliser la perspective *System Viewer*. Pour cela, il convient de cliquer sur *Open perspective* (① figure 38) puis *Other* (② figure 38) dans le cas où l'icône de passage à la perspective *System Viewer* n'est pas présent (③).



figure 38 Ouverture de perspective.

La fenêtre présentée figure 39 s'ouvre alors. Si la perspective *System Viewer* n'est pas présente, il faut cocher *Show all* (①). Sélectionner la perspective à utiliser (②), *System Viewer* en l'occurrence, puis cliquer *OK* (③).

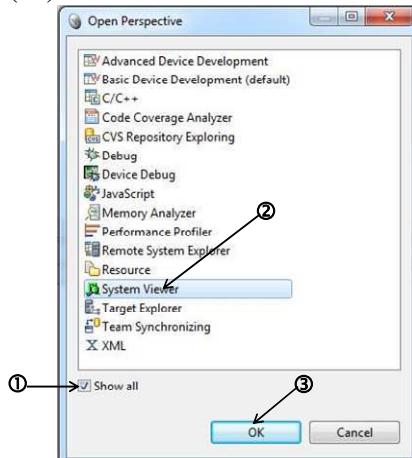


figure 39 Choix de la perspective à ouvrir.

La figure 40 montre le résultat de l'observation en utilisant la perspective *SystemViewer*. Il est possible d'obtenir la légende des pictogrammes utilisés pour l'affichage

en cliquant le bouton  (①).

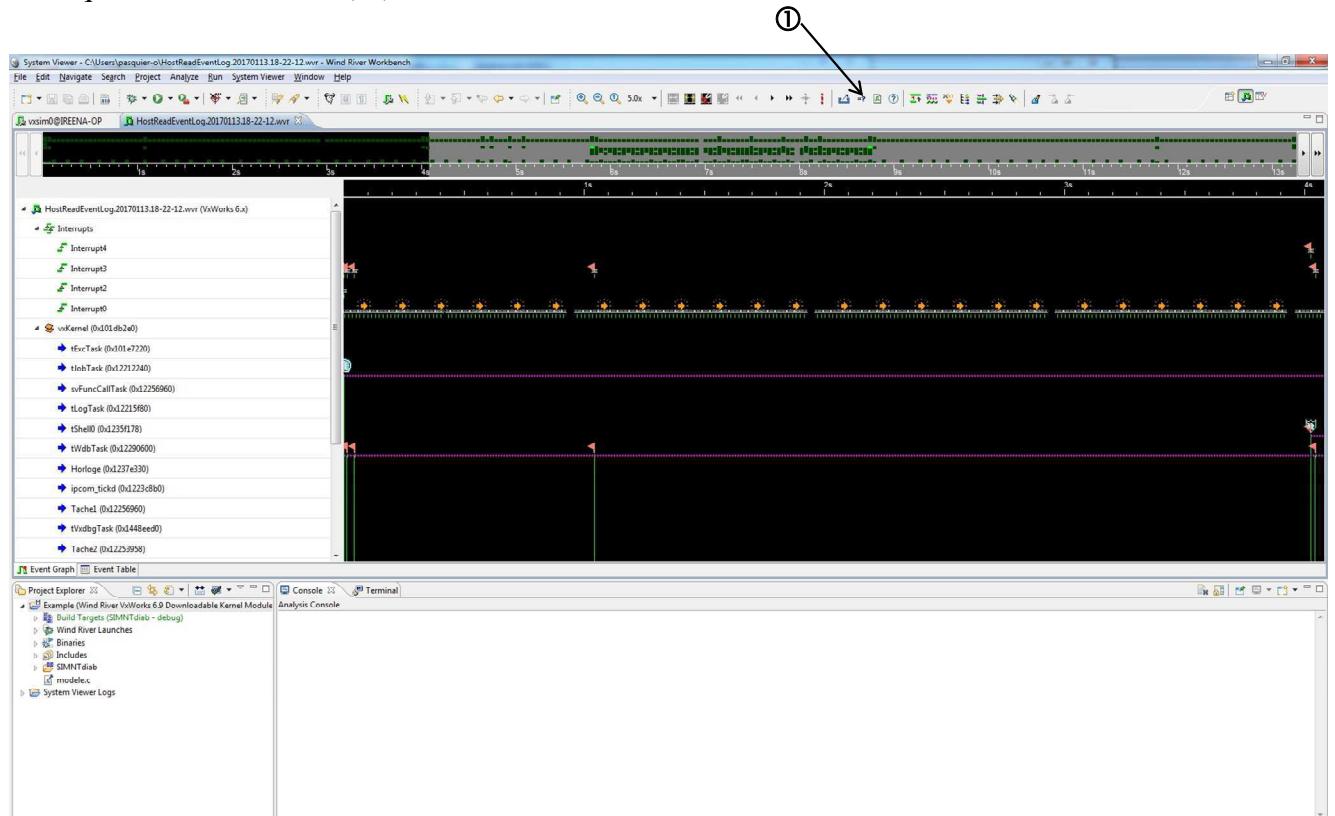


figure 40 Affichage des résultats d'observation avec la perspective System Viewer.

L'observation indique l'état de tous les éléments chargés sur la cible et de toutes les interruptions. Dans le cas des travaux pratiques ETN4, il est conseillé de limiter les éléments affichés à *Interrupt0* qui est l'interruption timer (utilisée par TaskDelay), les tâches de votre application et la tâche de fond (*Idle*). Pour cela, il faut, comme le montre la figure 41, sélectionner les éléments que l'on désire (①), puis cliquer bouton de droite et sélectionner *Hide Unselected* (②).

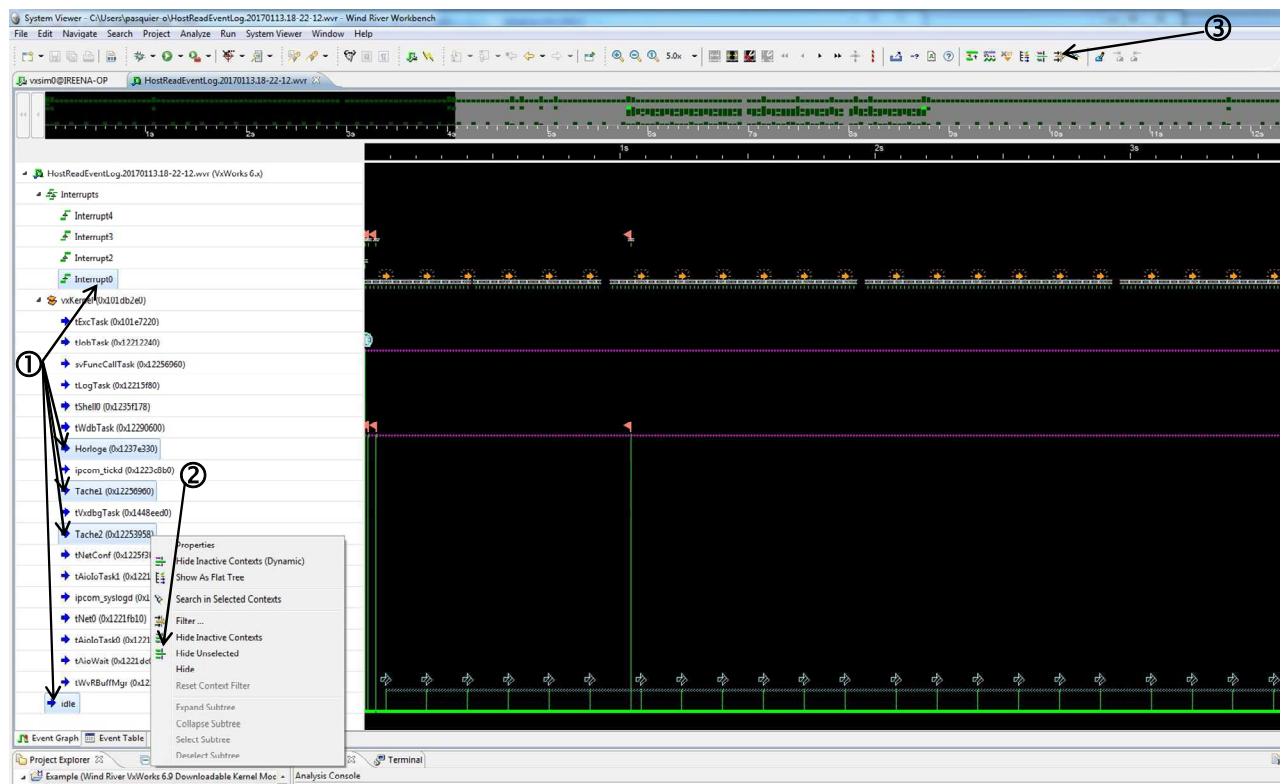


figure 41 Sélection des éléments à afficher.

La sélection des éléments que l'on souhaite observer peut aussi s'effectuer en utilisant le filtre d'affichage (bouton ③ figure 41). Cliquer sur ce bouton conduit à l'ouverture de la fenêtre présentée par la figure 42 dans laquelle il convient de sélectionner (cocher) les éléments que l'on souhaite faire apparaître lors de l'analyse des résultats.

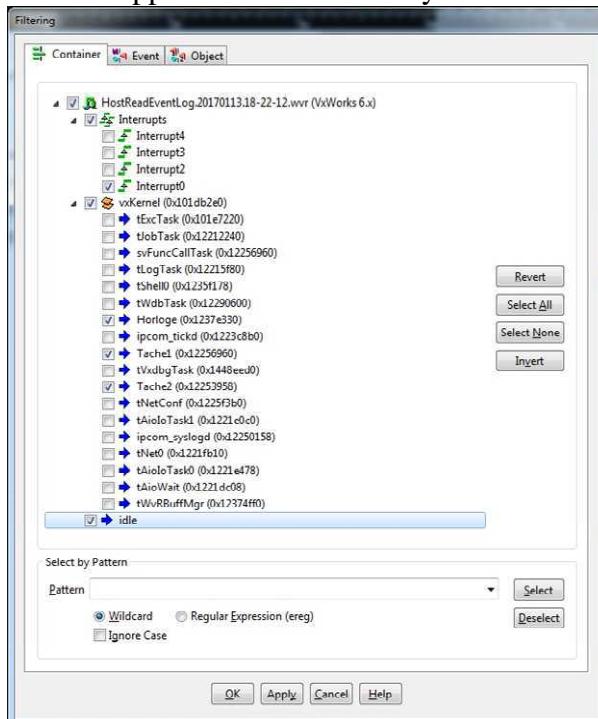


figure 42 Fenêtre de filtrage des éléments à afficher.

La figure 43 présente un exemple d'observation de l'application. *System Viewer*

indique schématiquement l'ensemble des événements observés dans une zone appelée « radar » (①) et en noir la zone affichée par rapport la totalité de l'observation (②). Il est possible de déplacer cette zone et de l'étirer ou encore d'utiliser des zooms (③). L'outil *System Viewer* permet aussi de nombreuses observations dont les mesures de temps.

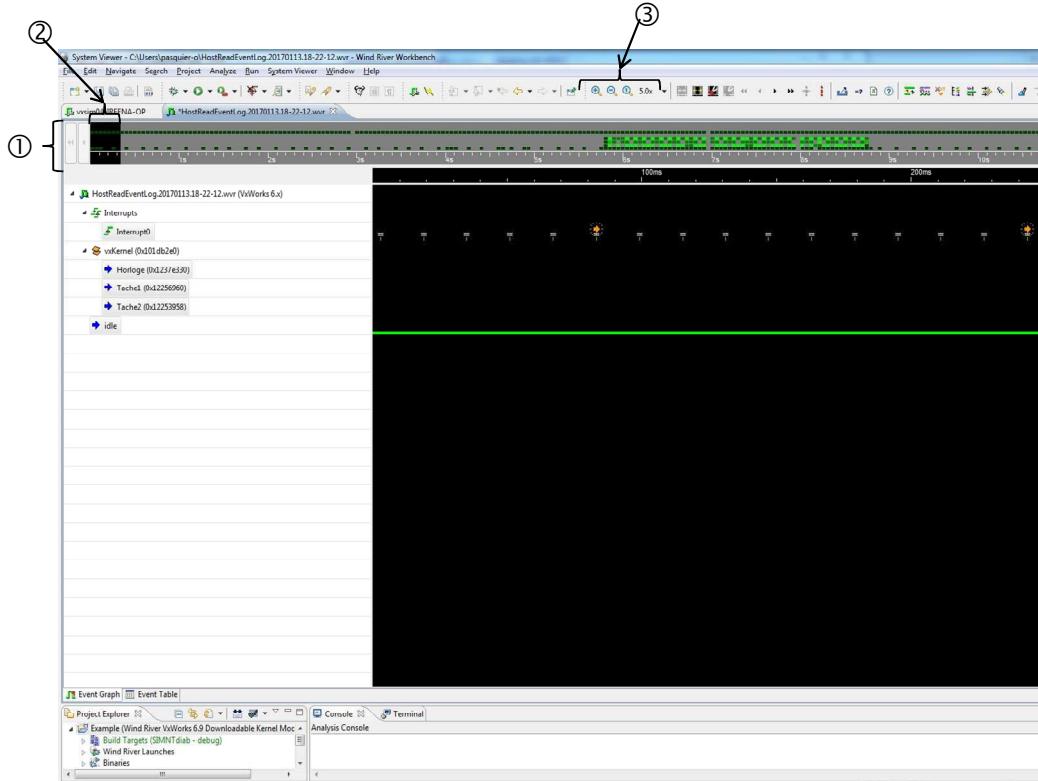


figure 43 Possibilités de SystemViewer.

La zone « radar » peut être utilisée pour repérer les zones intéressantes de la capture. Par exemple, la figure 44 présente une observation de l'application obtenue à partir de modèle.c. La zone ① montre une forte activité pour les tâches de l'application, c'est donc la zone qu'il est intéressant d'observer et plus particulièrement le début de cette zone (cas de la figure 44)

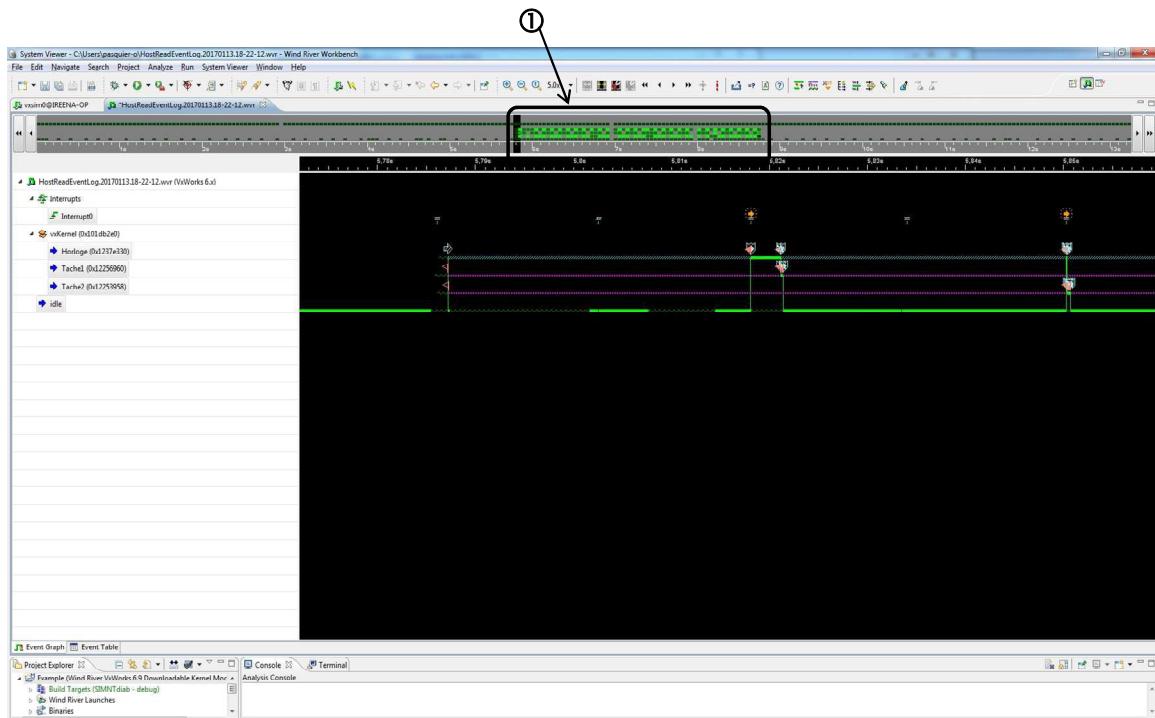


figure 44 Exemple d'observation avec System Viewer, utilisation de la zone radar.

Il existe une autre possibilité pour effectuer un zoom que celle présentée figure 43. Elle est présentée figure 45. Dans un premier temps, il faut délimiter la zone à zoomer en maintenant l'appui sur le bouton droit de la souris (①), puis en cliquant sur le bouton de zoom entre les extrémités définies (②).

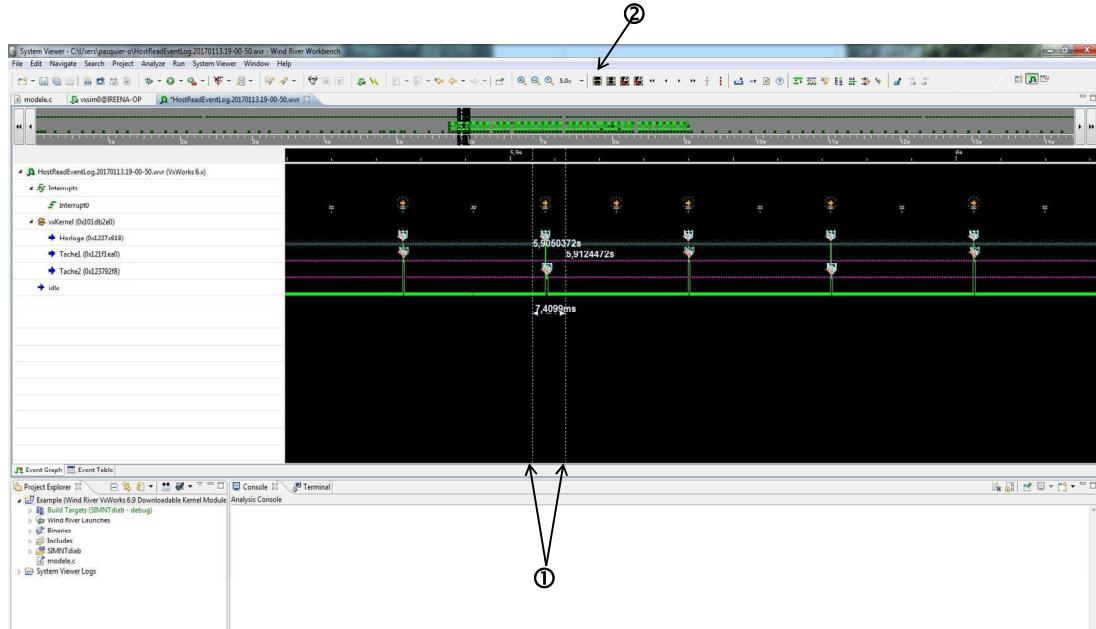


figure 45 Définition de la zone à zoomer.

La zone sélectionnée apparaît alors comme le montre la figure 46.

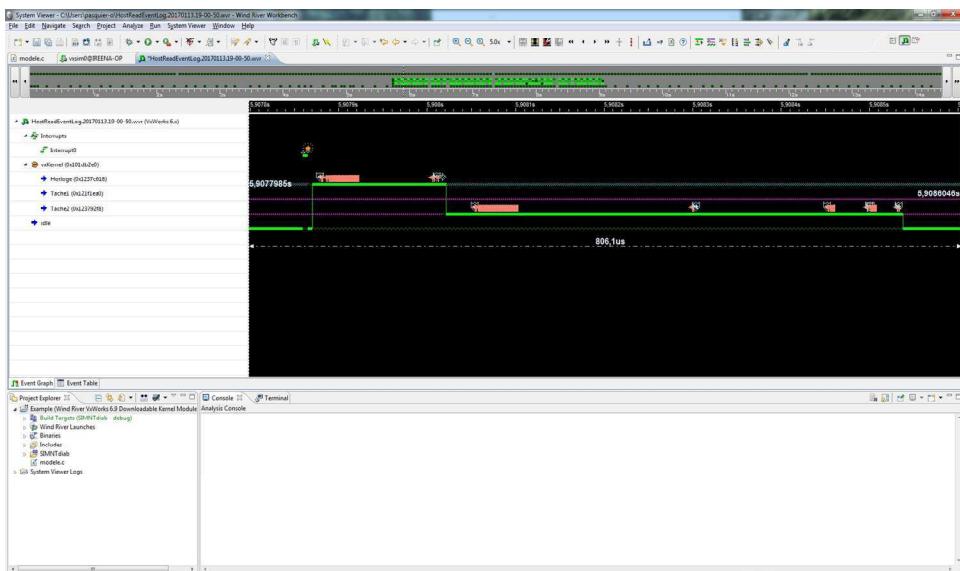


figure 46 Résultat d'un zoom.

Lorsque l'analyse du résultat est terminée, vous pouvez fermer la fenêtre d'observation. Il est aussi possible de sauvegarder la capture dans un fichier d'extension .wva (voir figure 47) en faisant *File* (①) puis *Save As* (②), cela amène à la procédure classique de sauvegarde d'informations dans un fichier. Ensuite, il est possible d'ouvrir l'observation à partir de Workbench (*File* (③) puis *Open File* (④)) comme pour un fichier classique en sélectionnant le bon type de fichier dans la fenêtre d'ouverture. Vous obtiendrez alors une fenêtre d'observation dans laquelle toutes les opérations sont possibles.

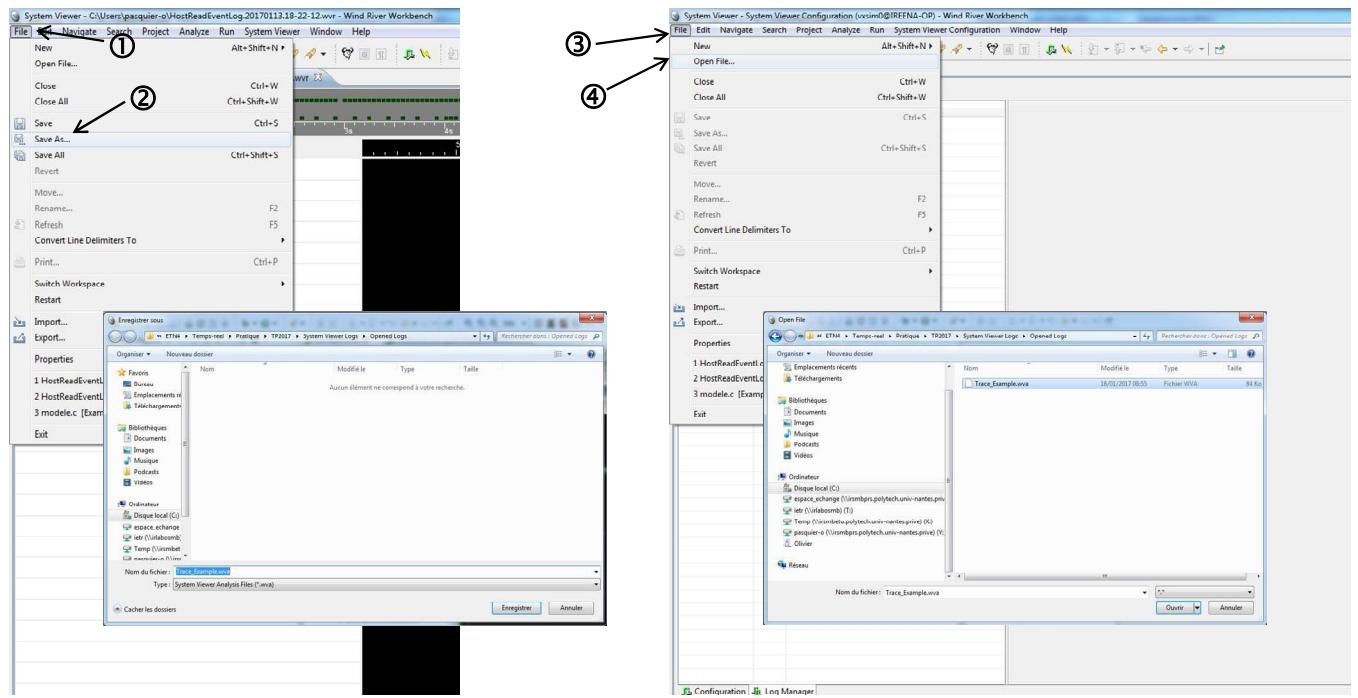


figure 47 Sauvegarde et ouverture d'un fichier d'observation.

Si, lors de l'observation avec System Viewer, vous observez plusieurs instances des éléments de votre application, le problème est lié à une mauvaise, ou à l'absence de, destruction des éléments de l'application avant rechargeement de celle-ci (procédure *Stop* de modèle.c). Il se peut alors que la destruction des éléments ne soit pas suffisante, il est



dans ce cas fortement conseillé de stopper totalement la cible et tous les outils qui lui sont liés, puis de tout redémarrer.

9 Mise au point d'une application

Les paragraphes précédents donnent les étapes à suivre pour créer et lancer une première fois une application. La mise au point de l'application nécessite ensuite de reproduire seulement certaines étapes si les outils démarrés n'ont pas été stoppés. Ces étapes sont les suivantes :

- Correction du fichier source,
- Compilation,
- Téléchargement,
- Démarrage de l'observation par *System Viewer*,
- Départ de l'application (fenêtre interface de la cible),
- Arrêt et destruction de l'application (fenêtre interface de la cible),
- Arrêt de l'observation par *System Viewer*,
- Récupération puis analyse des résultats de l'observation.

Après une observation avec *System Viewer*, il est possible de conserver cette observation et d'effectuer la séquence des opérations à partir de la modification du fichier source comme le montre la figure 48. Pour cela, il suffit de retourner à la perspective *Basic Device Development* (①).

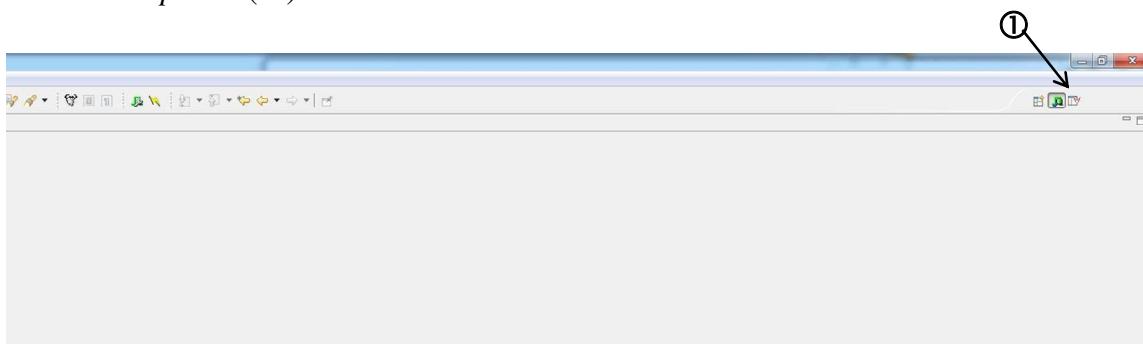


figure 48 Retour à la perspective Basic Device Development.

Il est alors possible de modifier le fichier source, de compiler le projet, de le télécharger sur le simulateur. Pour relancer l'observation, il faut sélectionner la fenêtre de configuration de *System Viewer* (① figure 49), relancer une observation (②) puis lancer et stopper l'application sur la cible et enfin stopper l'observation (③)



figure 49 Relance d'une observation par System Viewer.

10 Démarrage d'un simulateur spécifique

Il est possible de lancer un simulateur spécifique à partir du menu de la fenêtre *Target*

Manager (① figure 50) en utilisant le bouton . Le lancement du simulateur peut aussi être fait en cliquant bouton de droite dans la fenêtre *Remote Systems* puis *New* puis *Connection* (partie droite de figure 50).

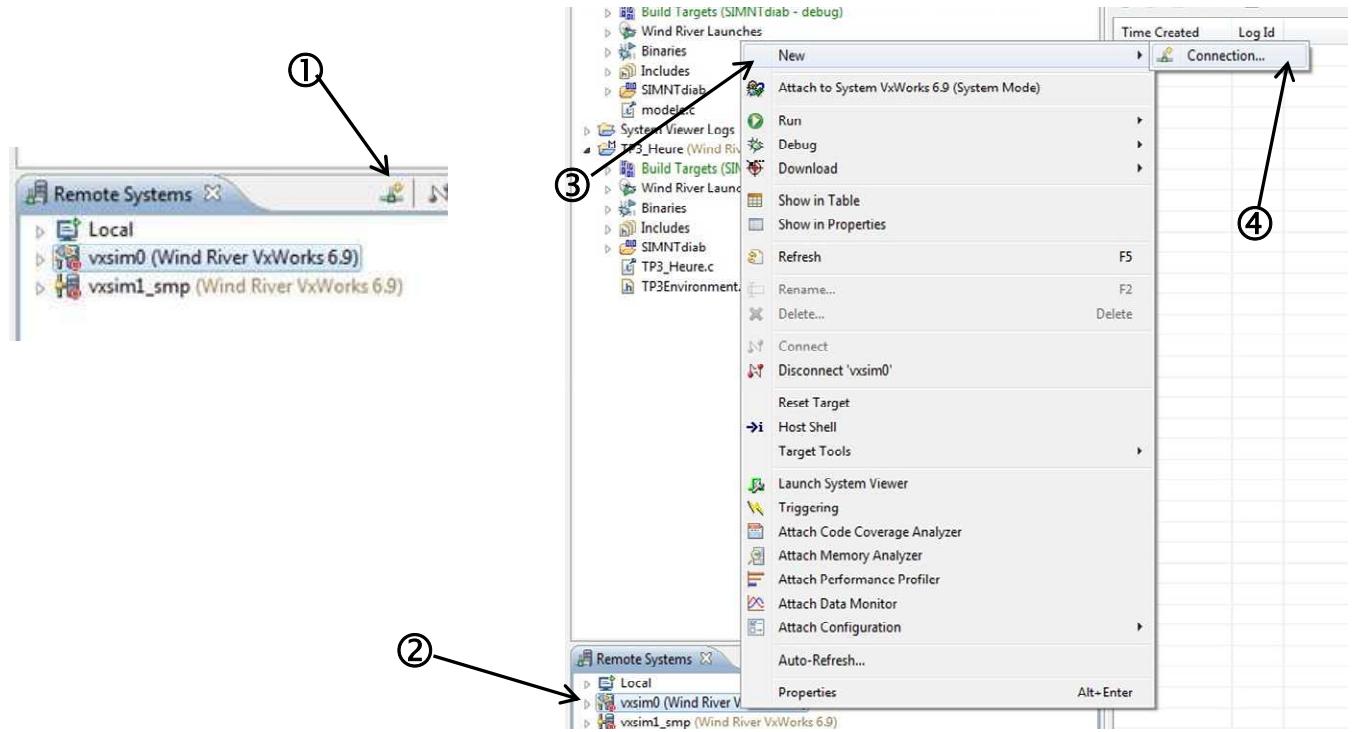


figure 50 Lancement d'un simulateur spécifique.

Un ensemble de fenêtres (figure 51) apparaissent alors pour définir le type de cible puis le type de simulateur et le type de connexion entre *Workbench* et le simulateur. La première (partie gauche de la figure 51) a pour objectif de déterminer le type de cible. Il faut sélectionner *Wind River VxWorks 6.x Simulator Connection* (①), puis passer à la fenêtre suivante par *next* (②). La seconde fenêtre a pour objectif de définir les paramètres de démarrage du simulateur (partie droite de la figure 51). Dans le cas d'un simulateur spécifique, il convient d'abord de sélectionner *Custom Simulator* (③), de cibler le code *vxWorks* qui définit les propriétés du simulateur. Les fenêtres suivantes ont pour objectif de déterminer la configuration de la connexion entre *Workbench* et le simulateur. Il faut ensuite désigner le fichier à utiliser pour le simulateur (④), la valeur donnée au champ *Processor Number* n'importe pas dans ce cas. Dans le cadre des Travaux Pratiques ETN4, il est possible de conserver tout le paramétrage par défaut et donc de terminer la configuration (⑤) dès la seconde fenêtre.

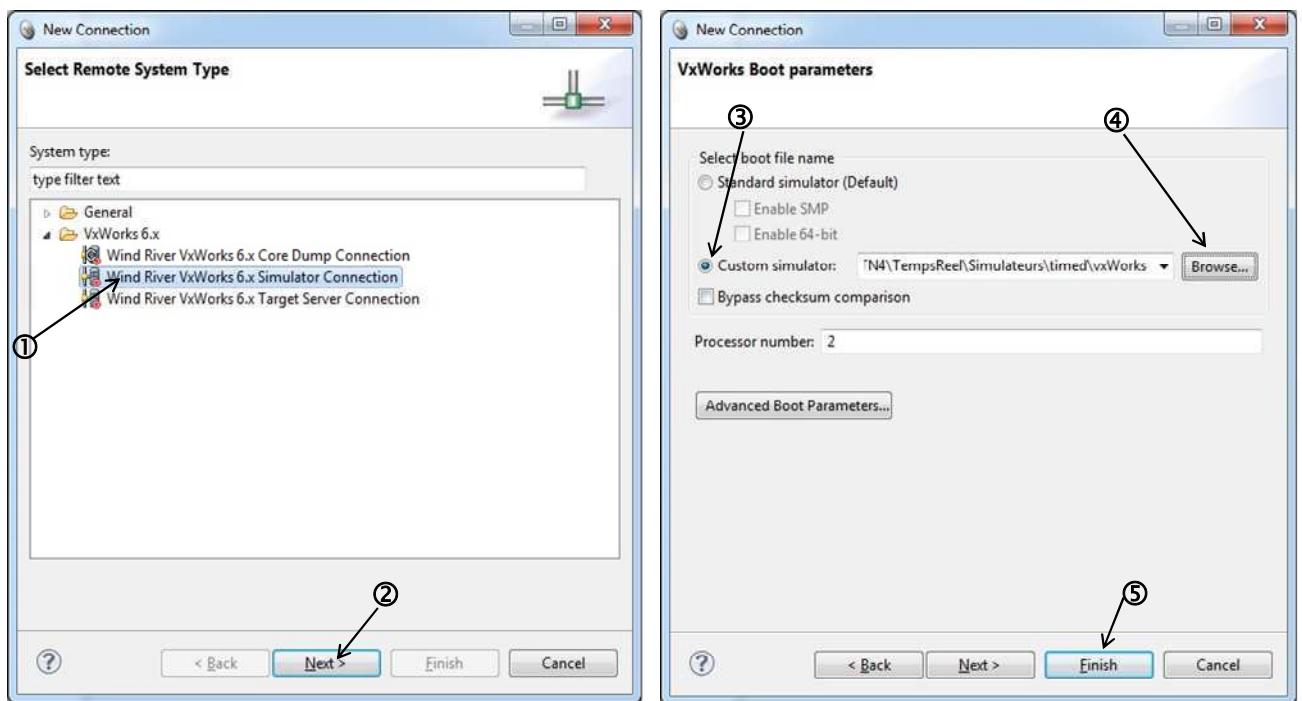


figure 51 Connexion et configuration d'un simulateur spécifique.

Une nouvelle fenêtre interface du simulateur est créée comparable à celle présentée par la figure 21 et une nouvelle ligne est ajoutée dans la fenêtre *Target Manager*.

Deux simulateurs spécifiques (fichiers vxworks) sont présents sur le volume `/ifsmbetu/temp` dans les répertoires `PROFESSEUR/Info_Indus_ETN/ETN4/Temps_Reel/simulateurs` puis les répertoires `timed` et `untimed`. Ils diffèrent par la façon de dater les résultats des observations. Le simulateur présent dans le répertoire `timed` associe à chaque événement sa date d'apparition alors que le simulateur présent dans le répertoire `untimed` associe à chaque événement son ordre d'apparition sans jamais associer de date. Le simulateur présent dans le répertoire `untimed` sera utilisé lors de la troisième manipulation.

11 Exemple de programme

Le programme ci-dessous (pages suivantes) permet d'implanter une application composée d'une tâche (horloge) qui éveille deux tâches (Tache1 et Tache2) par l'intermédiaire d'un sémaphore (*evt*) comme le montre la figure 52. Ces deux tâches partagent une variable *Var*.

Les procédures *start* et *stop* du programme donné dans les pages suivantes permettent respectivement de démarrer, c'est-à-dire créer et activer tous les éléments, et arrêter, c'est-à-dire détruire tous les éléments de l'application. Il est important de noter l'ordre des créations et des suppressions des éléments :

- Dans la procédure *start*, les relations sont créées et initialisées dans un premier temps, puis les tâches sont créées.
- Dans la procédure *stop*, les tâches sont détruites d'abord, puis seulement ensuite les relations.

Il est important de respecter cet ordre pour éviter tout aléa lors des phases de démarrage ou d'arrêt de l'application car cela garantit qu'à aucun moment une tâche n'accède à une relation dont l'organisation n'est pas déterministe (non initialisée ou déjà désallouée).

Le fichier *modele.c* contenant ce programme se trouve sur le volume <\\irsmbtu\\temp> dans le répertoire PROFESSEUR\Info_Indus_ETN\ETN4\TempsReel\modeles.

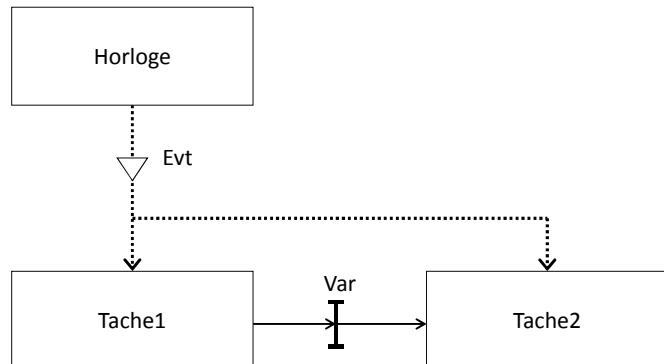


figure 52 Exemple d'application.

```

/* inclusion des bibliotheques*/
#include "vxWorks.h"
#include "stdio.h"
#include "semLib.h"
#include "taskLib.h"

#define STACK_SIZE (size_t)20000

/* declaration des semaphores */
SEM_ID evt;

/* declaration des Tid des taches */
TASK_ID tidHorloge;
TASK_ID tidTache1;
TASK_ID tidTache2;

/* declaration des variables partagees */
int Var;

/* declaration du code des taches */
void CodeHorloge(void)
{
    while(1)
    {
        taskDelay(2);
        printf("Horloge eveillée\n");
        semGive(evt);
    }
}

void CodeTache1(void)
{
    /* declaration de variables locales de la tache */
    int i;

    /* initialisation des variables locales */
    i = 1;
    while(1)
    {
        /* attente du semaphore d'eveil */
        semTake(evt, WAIT_FOREVER);
        printf("Tache1 modifie Var\n");
        Var = i;
        i = i + 10;
    }
};

void CodeTache2(void)
{
    /* declaration de variables locales de la tache */
    int i;

    while(1)

```

```

{
/* attente du semaphore d'eveil */
semTake(evt, WAIT_FOREVER);
i = Var;
printf("Tache2 a lu la valeur %d\n", i);
}
};

/* Procedure de demarrage de l'application */
int start()
{
/* Attention de bien créer les relations puis seulement ensuite les taches */

/* creation des semaphores */
evt = semBCreate(SEM_Q_FIFO, SEM_EMPTY);

/* demarrage des taches */
printf("demarrage de Tache1\n");
tidTache1 = taskSpawn("Tache1", 20, 0, STACK_SIZE, (FUNCPTR)CodeTache1,
0,0,0,0,0,0,0,0);
printf("demarrage de Decompteur\n");
tidTache2 = taskSpawn("Tache2", 25, 0, STACK_SIZE, (FUNCPTR)CodeTache2,
0,0,0,0,0,0,0,0);
printf("demarrage de horloge\n");
tidHorloge = taskSpawn("Horloge", 15, 0, STACK_SIZE, (FUNCPTR)CodeHorloge,
0,0,0,0,0,0,0,0);
printf("fin de start\n");
return (OK);
};

/* Procedure d'arrêt de l'application */
void stop()
{
/* Attention de bien détruire les taches puis seulement ensuite les relations */

printf("début de stop\n");
/* destruction des taches */
taskDelete(tidTache1);
taskDelete(tidTache2);
taskDelete(tidHorloge);
/* destruction du semaphore */
semDelete(evt);
printf("fin de stop\n");
}

```


Extrait de la documentation de VxWorks 6.0

Ce document présente une documentation succincte de quelques primitives de VxWorks 6.0. Une documentation détaillée peut être obtenue par l'aide en ligne de WorkBench. Les informations présentées dans ces quelques pages sont les principales caractéristiques de VxWorks qui permettent de réaliser et comprendre les travaux pratiques de 4^{ème} année. Selon les choix que vous ferez, toutes ne seront pas nécessaires. Il vous est tout de même conseillé de consulter au moins les rubriques suivantes :

- taskLib,
- taskSpawn,
- taskDelete,
- taskDelay
- semLib,
- semBLib,
- semMLib,
- semBCreate,
- semDelete
- semGive,
- semTake,
- msgQLib,
- msgQCreate,
- msgQDelete,
- msgQSend,
- msgQReceive

1 Gestion des tâches

1.1 **taskLib**

NAME

taskLib - task management library

ROUTINES

taskSpawn() - spawn a task
 taskCreate() - allocate and initialize a task without activation
 taskInitExcStk() - initialize a task with stacks at specified addresses
 taskInit() - initialize a task with a stack at a specified address
 taskActivate() - activate a task that has been initialized
 exit() - exit a task (ANSI)
 taskExit() - exit a task
 taskDelete() - delete a task
 taskDeleteForce() - delete a task without restriction
 taskSuspend() - suspend a task
 taskResume() - resume a task
 taskRestart() - restart a task
 taskPrioritySet() - change the priority of a task
 taskPriorityGet() - examine the priority of a task
 taskLock() - disable task rescheduling
 taskUnlock() - enable task rescheduling

taskSafe() - make the calling task safe from deletion
 taskUnsafe() - make the calling task unsafe from deletion
 taskDelay() - delay a task from executing
 taskIdSelf() - get the task ID of a running task
 taskIdVerify() - verify the existence of a task
 taskTcb() - get the task control block for a task ID

DESCRIPTION

This library provides the interface to the VxWorks task management facilities. Task control services are provided by the VxWorks kernel, which is comprised of kernelLib, taskLib, semLib, tickLib, msgQLib, and wdLib. Programmatic access to task information and debugging features is provided by taskInfo. Higher-level task information display routines are provided by taskShow.

TASK CREATION

Tasks are created with the general-purpose routine taskSpawn(). Task creation consists of the following: allocation of memory for the stack and task control block (WIND_TCB), initialization of the WIND_TCB, and activation of the WIND_TCB. Special needs may require the use of the lower-level routines taskInit() and taskActivate(), which are the underlying primitives of taskSpawn().

Tasks in VxWorks execute in the most privileged state of the underlying architecture. In a shared address space, processor privilege offers no protection advantages and actually hinders performance.

There is no limit to the number of tasks created in VxWorks, as long as sufficient memory is available to satisfy allocation requirements.

The routine sp() is provided in usrLib as a convenient abbreviation for spawning tasks. It calls taskSpawn() with default parameters.

TASK DELETION

If a task exits its "main" routine, specified during task creation, the kernel implicitly calls exit() to delete the task. Tasks can be explicitly deleted with the taskDelete() or exit() routine.

Task deletion must be handled with extreme care, due to the inherent difficulties of resource reclamation. Deleting a task that owns a critical resource can cripple the system, since the resource may no longer be available. Simply returning a resource to an available state is not a viable solution, since the system can make no assumption as to the state of a particular resource at the time a task is deleted.

The solution to the task deletion problem lies in deletion protection, rather than overly complex deletion facilities. Tasks may be protected from unexpected deletion using taskSafe() and taskUnsafe(). While a task is safe from deletion, deleters will block until it is safe to proceed. Also, a task can protect itself from deletion by taking a mutual-exclusion semaphore created with the SEM_DELETE_SAFE option, which enables an implicit taskSafe() with each semTake(), and a taskUnsafe() with each semGive() (see semMLib for more information). Many VxWorks system resources are protected in this manner, and application designers may wish to consider this facility where dynamic task deletion is a possibility.

The sigLib facility may also be used to allow a task to execute clean-up code before actually expiring.

TASK CONTROL

Tasks are manipulated by means of an ID that is returned when a task is created. VxWorks uses the convention that specifying a task ID of NULL in a task control function signifies the calling task.

The following routines control task state: taskResume(), taskSuspend(), taskStop(), taskCont(), taskDelay(), taskRestart(), taskPrioritySet(), and taskRegsSet().

TASK SCHEDULING

VxWorks schedules tasks on the basis of priority. Tasks may have priorities ranging from 0, the highest priority, to 255, the lowest priority. The priority of a task in VxWorks is dynamic, and an existing task's priority can be changed using taskPrioritySet().

INCLUDE FILES

taskLib.h

SEE ALSOtaskInfo, taskShow, taskHookLib, taskVarLib, semLib, semMLib, kernelLib, VxWorks
Programmer's Guide: Basic OS**1.2 taskActivate()****NAME**

taskActivate() - activate a task that has been initialized

SYNOPSISSTATUS taskActivate
(
int tid /* task ID of task to activate */
)**DESCRIPTION**

This routine activates tasks created by taskInit(). Without activation, a task is ineligible for CPU allocation by the scheduler.

The tid (task ID) argument is simply the address of the WIND_TCB for the task (the taskInit() pTcb argument), cast to an integer:

tid = (int) pTcb;

The taskSpawn() routine is built from taskActivate() and taskInit(). Tasks created by taskSpawn() do not require explicit task activation.

RETURNS

OK, or ERROR if the task cannot be activated.

ERRNO

S_objLib_OBJ_ID_ERROR

The tid parameter is an invalid task ID.

SEE ALSO

taskLib, taskInit()

1.3 taskDelay()**NAME**

taskDelay() - delay a task from executing

SYNOPSISSTATUS taskDelay
(
int ticks /* number of ticks to delay task */
)**DESCRIPTION**

This routine causes the calling task to relinquish the CPU for the duration specified (in ticks). This is commonly referred to as manual rescheduling, but it is also useful when waiting for some external condition that does not have an interrupt associated with it.

If the calling task receives a signal that is not being blocked or ignored, taskDelay() returns ERROR and sets errno to EINTR after the signal handler is run.

RETURNS

OK, or ERROR if called from interrupt level or if the calling task receives a signal that is not blocked or ignored.

ERRNO

S_intLib_NOT_ISR_CALLABLE
EINTR

SEE ALSO
taskLib

1.4 **taskDelete()**

NAME

taskDelete() - delete a task

SYNOPSIS

```
STATUS taskDelete
(
    int tid /* task ID of task to delete */
)
```

DESCRIPTION

This routine causes a specified task to cease to exist and deallocates the stack and WIND_TCB memory resources. Upon deletion, all routines specified by taskDeleteHookAdd() will be called in the context of the deleting task. This routine is the companion routine to taskSpawn().

WARNING

Deleting individual user tasks, as opposed to deleting the entire RTP, may result in unpredictable RTP behavior. The deletion of individual user tasks should only be performed for debugging purposes.

RETURNS

OK, or ERROR if the task cannot be deleted.

ERRNO

S_intLib_NOT_ISR_CALLABLE
S_objLib_OBJ_DELETED
S_objLib_OBJ_UNAVAILABLE
S_objLib_OBJ_ID_ERROR

SEE ALSO

taskLib, excLib, taskDeleteHookAdd(), taskSpawn(),
VxWorks Programmer's Guide: Basic OS

1.5 **TaskLock, TaskUnlock**

The **taskLock()** routine disables preemption of the calling task by other tasks. The calling task is the only task that is allowed to execute--as long as it is in the ready state. If the calling task blocks or suspends, the scheduler selects the next highest-priority eligible task to execute. When the calling task unblocks and resumes execution, preemption is again disabled. (For information about task states, see [6.2.1 Task States and Transitions](#).) The **taskUnlock()** routine reenables preemption by other tasks.

The task lock routines are used to protect a critical region of code as follow:

```
foo()
{
    taskLock();
    .
    . /* critical region of code that cannot be interrupted */
    .
    taskUnlock();
}
```

Task locks can be nested (they are implemented using a count variable), in which case preemption is not reenabled until **taskUnlock()** has been called as many times as **taskLock()**.

Task locks can lead to unacceptable real-time response times. Tasks of higher priority are unable to execute until the locking task leaves the critical region, even though the higher-priority task is not itself involved with the critical region. While this kind of mutual exclusion is simple, be sure to keep the duration short. In general, semaphores provide a better mechanism for mutual exclusion; see [7.6 Semaphores](#).

Interrupts are not blocked when **taskLock()** is used, but it can be paired with **intLock()** to disable preemption by tasks and ISRs.

For more information, see the VxWorks API reference entries for **taskLock()** and **taskUnLock()**.

NOTE: The **taskLock()** and **taskUnlock()** routines are provided for the UP configuration of VxWorks, but not the SMP configuration. Several alternatives are available for SMP systems, including task-only spinlocks, which default to **taskLock()** and **taskUnlock()** behavior in a UP system. For more information, see [23.7.2 Task-Only Spinlocks](#) and [23.17 Migrating Code to VxWorks SMP](#).

1.6 **taskIdSelf()**

NAME

`taskIdSelf()` - get the task ID of a running task

SYNOPSIS

```
int taskIdSelf (void)
```

DESCRIPTION

This routine gets the task ID of the calling task. The task ID will be invalid if called at interrupt level.

RETURNS

The task ID of the calling task.

ERRNO

N/A

SEE ALSO

`taskLib`

1.7 **taskInit()**

NAME

taskInit() - initialize a task with a stack at a specified address

SYNOPSIS

STATUS taskInit

```
(  
    FAST WIND_TCB *pTcb,    /* address of new task's TCB */  
    char *      name,      /* name of new task (stored at pStackBase) */  
    int        priority,   /* priority of new task */  
    int        options,    /* task option word */  
    char *      pStackBase, /* base of new task's execution stack */  
    int        stackSize,  /* size (bytes) of stack needed */  
    FUNCPTR    entryPt,   /* entry point of new task */  
    int        arg1,      /* 1st of 10 req'd args to pass to entryPt */  
    int        arg2,  
    int        arg3,  
    int        arg4,  
    int        arg5,  
    int        arg6,  
    int        arg7,  
    int        arg8,  
    int        arg9,  
    int        arg10  
)
```

DESCRIPTION

This routine initializes user-specified regions of memory for a task stack and control block instead of allocating them from memory as taskSpawn() does. This routine will utilize the specified pointers to the WIND_TCB and stack as the components of the task. This allows, for example, the initialization of a static WIND_TCB variable. It also allows for special stack positioning as a debugging aid.

As in taskSpawn(), a task may be given a name. While taskSpawn() automatically names unnamed tasks, taskInit() permits the existence of tasks without names. The task ID required by other task routines is simply the address pTcb, cast to an integer.

Unlike taskSpawn(), taskInit() allows one to control the activation of the VX DEALLOC_STACK bit in options; taskSpawn() always sets this bit. Setting this bit causes the stack to be automatically deallocated when a taskDelete() is performed, or when a return command is issued from the entry function. If a pre-allocated stack is used, the VX DEALLOC_STACK bit must not be set.

The pStackBase parameter specifies the base of the execution stack. The stack may grow up or down from pStackBase depending on the target architecture. The caller is responsible for setting up any guard zones around the specified stack area. The following code fragment illustrates how to specify the stack base location:

For architectures where the stack grows down:

```
pStackMem = (char *) malloc (stackSize);  
  
if (pStackMem != NULL)  
    status = taskInit ( ... , pStackMem + stackSize, stackSize, ... );
```

For architectures where the stack grows up:

```
pStackMem = (char *) malloc (stackSize);  
  
if (pStackMem != NULL)  
    status = taskInit ( ... , pStackMem, stackSize, ... );
```

Please note that malloc() is used in the above code fragment for illustrative purposes only since it's a well-known API. Typically, the stack memory would be obtained by some other mechanism.

Other arguments are the same as in taskSpawn(). Unlike taskSpawn(), taskInit() does not activate the task. This must be done by calling taskActivate() after calling taskInit().

Normally, tasks should be started using taskSpawn() rather than taskInit(), except when additional control is required for task memory allocation or a separate task activation is desired.

NOTE

For future releases of VxWorks, each task has an exception stack. To continue providing the taskInit() API, this routine now carves memory for the exception stack of a task. To use a specific region of memory for the exception stack, use the routine taskInitExcStk() instead.

RETURNS

OK, or ERROR if the task cannot be initialized.

ERRNO

S_intLib_NOT_ISR_CALLABLE
S_objLib_OBJ_ID_ERROR
S_taskLib_ILLEGAL_PRIORITY

SEE ALSO

taskLib, taskActivate(), taskSpawn(), taskInitExcStk()

1.8 **taskLock()**

NAME

taskLock() - disable task rescheduling

SYNOPSIS

STATUS taskLock (void)

DESCRIPTION

This routine disables task context switching. The task that calls this routine will be the only task that is allowed to execute, unless the task explicitly gives up the CPU by making itself no longer ready. Typically this call is paired with taskUnlock(); together they surround a critical section of code. These preemption locks are implemented with a counting variable that allows nested preemption locks. Preemption will not be unlocked until taskUnlock() has been called as many times as taskLock().

This routine does not lock out interrupts; use intLock() to lock out interrupts.

A taskLock() is preferable to intLock() as a means of mutual exclusion, because interrupt lock-outs add interrupt latency to the system.

A semTake() is preferable to taskLock() as a means of mutual exclusion, because preemption lock-outs add preemptive latency to the system.

The taskLock() routine is not callable from interrupt service routines.

RETURNS

OK or ERROR.

ERRNO

S_objLib_OBJ_ID_ERROR
S_intLib_NOT_ISR_CALLABLE

SEE ALSO

taskLib, taskUnlock(), intLock(), taskSafe(), semTake()

1.9 **taskPriorityGet()**

NAME

taskPriorityGet() - examine the priority of a task

SYNOPSIS

STATUS taskPriorityGet

```

(
  int tid,    /* task ID */
  int * pPriority /* return priority here */
)

```

DESCRIPTION

This routine determines the current priority of a specified task. The current priority is copied to the integer pointed to by pPriority.

RETURNS

OK, or ERROR if the task ID is invalid.

ERRNO

S_objLib_OBJ_ID_ERROR

SEE ALSO

taskLib, taskPrioritySet()

1.10 taskPrioritySet()**NAME**

taskPrioritySet() - change the priority of a task

SYNOPSIS

```

STATUS taskPrioritySet
(
  int tid,    /* task ID */
  int newPriority /* new priority */
)

```

DESCRIPTION

This routine changes a task's priority to a specified priority. Priorities range from 0, the highest priority, to 255, the lowest priority.

A request to lower the priority of a task that has acquired a priority inversion safe mutex semaphore will not take immediate effect. To prevent a priority inversion situation, the requested lower priority will take effect, in general, only after the task relinquishes all priority inversion safe mutex semaphores.

A request to raise the priority of a task will take immediate effect.

RETURNS

OK, or ERROR if the task ID is invalid.

ERRNO

S_taskLib_ILLEGAL_PRIORITY

S_objLib_OBJ_ID_ERROR

SEE ALSO

taskLib, taskPriorityGet()

1.11 taskResume()**NAME**

taskResume() - resume a task

SYNOPSIS

```

STATUS taskResume
(
  int tid /* task ID of task to resume */
)

```

DESCRIPTION

This routine resumes a specified task. Suspension is cleared, and the task operates in the remaining state.

RETURNS

OK, or ERROR if the task cannot be resumed.

ERRNO

S_objLib_OBJ_ID_ERROR

SEE ALSO

taskLib

1.12 taskSpawn()**NAME**

taskSpawn() - spawn a task

SYNOPSIS

```
int taskSpawn
(
    char * name,    /* name of new task (stored at pStackBase) */
    int priority,  /* priority of new task */
    int options,   /* task option word */
    int stackSize, /* size (bytes) of stack needed plus name */
    FUNCPTR entryPt, /* entry point of new task */
    int arg1,     /* 1st of 10 req'd args to pass to entryPt */
    int arg2,
    int arg3,
    int arg4,
    int arg5,
    int arg6,
    int arg7,
    int arg8,
    int arg9,
    int arg10
)
```

DESCRIPTION

This routine creates and activates a new task with a specified priority and options and returns a system-assigned ID. See taskInit() and taskActivate() for the building blocks of this routine.

A task may be assigned a name as a debugging aid. This name will appear in displays generated by various system information facilities such as i(). The name may be of arbitrary length and content, but the current VxWorks convention is to limit task names to ten characters and prefix them with a "t". If name is specified as NULL, an ASCII name will be assigned to the task of the form "tn" where n is an integer which increments as new tasks are spawned.

VxWorks schedules tasks on the basis of priority. Tasks may have priorities ranging from 0, the highest priority, to 255, the lowest priority. The priority of a task in VxWorks is dynamic and one may change an existing task's priority with taskPrioritySet().

The only resource allocated to a spawned task is a stack of a specified size stackSize, which is allocated from the system memory partition. Stack size should be an even integer. A task control block (TCB) is carved from the stack, as well as any memory required by the task name. The remaining memory is the task's stack and every byte is filled with the value 0xEE for the checkStack() facility. See the manual entry for checkStack() for stack-size checking aids.

The entry address entryPt is the address of the "main" routine of the task. The routine will be called once the C environment has been set up. The specified routine will be called with the ten given arguments. Should the specified main routine return, a call to exit() will automatically be made.

Note that ten (and only ten) arguments must be passed for the spawned function.

Bits in the options argument may be set to run with the following modes:

VX_FP_TASK

execute with floating-point coprocessor support. A task which performs floating point operations or calls any functions which either return or take a floating point value as arguments must be created with this option. Some routines perform floating point operations internally. The VxWorks documentation for these clearly state the need to use the VX_FP_TASK option.

VX_ALTIVEC_TASK

execute with Altivec support (PowerPC only)

VX_SPE_TASK

execute with SPE support (PowerPC only)

VX_SPE_TASK

execute with SPE support (PowerPC only)

VX_DSP_TASK

execute with DSP support (SuperH only)

VX_PRIVATE_ENV

include private environment support (see envLib).

VX_NO_STACK_FILL

do not fill the stack for use by checkStack().

VX_UNBREAKABLE

do not allow breakpoint debugging.

VX_NO_STACK_PROTECT

do not provide stack protection: no overflow or underflow detection, stack remains executable.

See the definitions in taskLib.h.

RETURNS

The task ID, or ERROR if memory is insufficient or the task cannot be created.

ERRNO

S_intLib_NOT_ISR_CALLABLE

S_objLib_OBJ_ID_ERROR

S_smObjLib_NOT_INITIALIZED

S_memLib_NOT_ENOUGH_MEMORY

S_memLib_BLOCK_ERROR

S_taskLib_ILLEGAL_PRIORITY

SEE ALSO

taskLib, taskInit(), taskActivate(), sp(), VxWorks Programmer's Guide: Basic OS

1.13 taskSuspend()

NAME

taskSuspend() - suspend a task

SYNOPSIS

STATUS taskSuspend

(

int tid /* task ID of task to suspend */

)

DESCRIPTION

This routine suspends a specified task. A task ID of zero results in the suspension of the calling task. Suspension is additive, thus tasks can be delayed and suspended, or pended and suspended. Suspended, delayed tasks whose delays expire remain suspended. Likewise, suspended, pended tasks that unblock remain suspended only.

Care should be taken with asynchronous use of this facility. The specified task is suspended regardless of its current state. The task could, for instance, have mutual exclusion to some system resource, such as the network or system memory partition. If suspended during such a time, the facilities engaged are unavailable, and the situation often ends in deadlock.

This routine is the basis of the debugging and exception handling packages. However, as a synchronization mechanism, this facility should be rejected in favor of the more general semaphore facility.

RETURNS

OK, or ERROR if the task cannot be suspended.

ERRNO

S_objLib_OBJ_ID_ERROR

SEE ALSO

taskLib

taskUnlock()

NAME

taskUnlock() - enable task rescheduling

SYNOPSIS

STATUS taskUnlock (void)

DESCRIPTION

This routine decrements the preemption lock count. Typically this call is paired with taskLock() and concludes a critical section of code. Preemption will not be unlocked until taskUnlock() has been called as many times as taskLock(). When the lock count is decremented to zero, any tasks that were eligible to preempt the current task will execute.

The taskUnlock() routine is not callable from interrupt service routines.

RETURNS

OK or ERROR.

ERRNO

S_intLib_NOT_ISR_CALLABLE

SEE ALSO

taskLib, taskLock()

2 Gestion des semaphores

2.1 *semLib*

NAME

semLib - general semaphore library

ROUTINES

semGive() - give a semaphore

semTake() - take a semaphore

semFlush() - unblock every task pended on a semaphore

semDelete() - delete a semaphore

DESCRIPTION

Semaphores are the basis for synchronization and mutual exclusion in VxWorks. They are powerful in their simplicity and form the foundation for numerous VxWorks facilities.

Different semaphore types serve different needs, and while the behavior of the types differs, their basic interface is the same. This library provides semaphore routines common to all VxWorks semaphore types. For all types, the two basic operations are semTake() and semGive(), the acquisition or relinquishing of a semaphore.

Semaphore creation and initialization is handled by other libraries, depending on the type of semaphore used. These libraries contain full functional descriptions of the semaphore types:

semBLib - binary semaphores

semCLib - counting semaphores

semMLib - mutual exclusion semaphores

semSmLib - shared memory semaphores

Binary semaphores offer the greatest speed and the broadest applicability.

The semLib library provides all other semaphore operations, including routines for semaphore control, deletion, and information. Semaphores must be validated before any semaphore operation can be undertaken. An invalid semaphore ID results in ERROR, and an appropriate errno is set.

SEMAPHORE CONTROL

The semTake() call acquires a specified semaphore, blocking the calling task or making the semaphore unavailable. All semaphore types support a timeout on the semTake() operation. The timeout is specified as the number of ticks to remain blocked on the semaphore. Timeouts of WAIT_FOREVER and NO_WAIT codify common timeouts. If a semTake() times out, it returns ERROR. Refer to the library of the specific semaphore type for the exact behavior of this operation.

The semGive() call relinquishes a specified semaphore, unblocking a pended task or making the semaphore available. Refer to the library of the specific semaphore type for the exact behavior of this operation.

The semFlush() call may be used to atomically unblock all tasks pended on a semaphore queue, i.e., all tasks will be unblocked before any are allowed to run. It may be thought of as a broadcast operation in synchronization applications. The state of the semaphore is unchanged by the use of semFlush(); it is not analogous to semGive().

SEMAPHORE DELETION

The semDelete() call terminates a semaphore and deallocates any associated memory. The deletion of a semaphore unblocks tasks pended on that semaphore; the routines which were pended return ERROR. Take care when deleting semaphores, particularly those used for mutual exclusion, to avoid deleting a semaphore out from under a task that already has taken (owns) that semaphore. Applications should adopt the protocol of only deleting semaphores that the deleting task has successfully taken.

SEMAPHORE INFORMATION

The semInfo() call is a useful debugging aid, reporting all tasks blocked on a specified semaphore. It provides a snapshot of the queue at the time of the call, but because semaphores are dynamic, the information may be out of date by the time it is available. As with the current state of the semaphore, use of the queue of pended tasks should be restricted to debugging uses only.

VXWORKS EVENTS

If a task has registered for receiving events with a semaphore, events will be sent when that semaphore becomes available. By becoming available, it is implied that there is a change of state. For a binary semaphore, there is only a change of state when a semGive() is done on a semaphore that was taken. For a counting semaphore, there is always a change of state when the semaphore is available, since the count is incremented each time. For a mutex, a semGive() can only be performed if the current task is the owner, implying that the semaphore has been taken; thus, there is always a change of state.

INCLUDE FILES

semLib.h

SEE ALSO

taskLib, semBLib, semCLib, semMLib, semSmLib, semEvLib, eventLib, VxWorks
Programmer's Guide: Basic OS

2.2 **semGive()**

NAME

semGive() - give a semaphore

SYNOPSIS

```
STATUS semGive
(
    SEM_ID semId /* semaphore ID to give */
)
```

DESCRIPTION

This routine performs the give operation on a specified semaphore. Depending on the type of semaphore, the state of the semaphore and of the pending tasks may be affected. If no tasks are pending on the semaphore and a task has previously registered to receive events from the semaphore, these events are sent in the context of this call. This may result in the unpending of the task waiting for the events. If the semaphore fails to send events and if it was created using the SEM_EVENTSEND_ERR_NOTIFY option, ERROR is returned even though the give operation was successful. The behavior of semGive() is discussed fully in the library description of the specific semaphore type being used.

RETURNS

OK on success or ERROR otherwise

ERRNO

S_intLib_NOT_ISR_CALLABLE

Routine was called from an ISR for a mutex semaphore.

S_objLib_OBJ_ID_ERROR

Semaphore ID is invalid.

S_semLib_INVALID_OPERATION

Current task not owner of mutex semaphore.

S_semLib_COUNT_OVERFLOW

Counting semaphore was given when count was already at maximum.

S_eventLib_EVENTSEND_FAILED

Semaphore failed to send events to the registered task. This errno value can only exist if the semaphore was created with the SEM_EVENTSEND_ERR_NOTIFY option.

SEE ALSO

semLib, semBLib, semCLib, semMLib, semSmLib, semEvStart

2.3 *semTake()***NAME**

semTake() - take a semaphore

SYNOPSIS

```
STATUS semTake
(
    SEM_ID semId, /* semaphore ID to take */
    int    timeout /* timeout in ticks */
)
```

DESCRIPTION

This routine performs the take operation on a specified semaphore. Depending on the type of semaphore, the state of the semaphore and the calling task may be affected. The behavior of semTake() is discussed fully in the library description of the specific semaphore type being used.

A timeout in ticks may be specified. If a task times out, semTake() will return ERROR.

Timeouts of WAIT_FOREVER (-1) and NO_WAIT (0) indicate to wait indefinitely or not to wait at all.

When semTake() returns due to timeout, it sets the errno to S_objLib_OBJ_TIMEOUT (defined in objLib.h).

The semTake() routine is not callable from interrupt service routines.

RETURNS

OK, or ERROR if the semaphore ID is invalid or the task timed out.

ERRNO

S_intLib_NOT_ISR_CALLABLE

Routine was called from an ISR.

S_objLib_OBJ_ID_ERROR

Semaphore ID is invalid.
 S_objLib_OBJ_TIMEOUT
 Timeout occurred while pending on semaphore.
 S_objLib_OBJ_UNAVAILABLE
 Would have blocked but NO_WAIT was specified.

SEE ALSO

semLib, semBLib, semCLib, semMLib, semSmLib

2.4 semFlush()**NAME**

semFlush() - unblock every task pended on a semaphore

SYNOPSIS

STATUS semFlush
 (
 SEM_ID semId /* semaphore ID to unblock everyone for */
)

DESCRIPTION

This routine atomically unblocks all tasks pended on a specified semaphore, i.e., all tasks will be unblocked before any is allowed to run. The state of the underlying semaphore is unchanged. All pended tasks will enter the ready queue before having a chance to execute.

The flush operation is useful as a means of broadcast in synchronization applications. Its use is illegal for mutual-exclusion semaphores created with semMCreate().

RETURNS

OK, or ERROR if the semaphore ID is invalid or the operation is not supported.

ERRNO

S_objLib_OBJ_ID_ERROR

SEE ALSO

semLib, semBLib, semCLib, semMLib, semSmLib

API Reference: Routines

2.5 semDelete()**NAME**

semDelete() - delete a semaphore

SYNOPSIS

STATUS semDelete
 (
 SEM_ID semId /* semaphore ID to delete */
)

DESCRIPTION

This routine terminates and deallocates any memory associated with a specified semaphore. All tasks pending on the semaphore or pending for the reception of events meant to be sent from the semaphore will unblock and return ERROR.

WARNING

Take care when deleting semaphores, particularly those used for mutual exclusion, to avoid deleting a semaphore out from under a task that already has taken (owns) that semaphore. Applications should adopt the protocol of only deleting semaphores that the deleting task has successfully taken.

RETURNS

OK, or ERROR if the semaphore ID is invalid.

ERRNO

S_intLib_NOT_ISR_CALLABLE
 Routine cannot be called from ISR.
 S_objLib_OBJ_ID_ERROR
 Semaphore ID is invalid.
 S_smObjLib_NO_OBJECT_DESTROY
 Deleting a shared semaphore is not permitted
 S_objLib_OBJ_OPERATION_UNSUPPORTED
 Deleting a named semaphore is not permitted.
 SEE ALSO
 semLib, semBLib, semCLib, semMLib, semSmLib

2.6 semInit()**NAME**

semInit() - initialize a static binary semaphore

SYNOPSIS

```

STATUS semInit
(
  SEMAPHORE *pSemaphore /* 4.x semaphore to initialize */
)

```

DESCRIPTION

This routine initializes static VxWorks 4.x semaphores. In some instances, a semaphore cannot be created with semCreate() but is a static object.

RETURNS

OK, or ERROR if the semaphore cannot be initialized.

ERRNO

N/A

SEE ALSO

semOLib, semCreate()
 API Reference: Routines

2.7 semClear()**NAME**

semClear() - take a release 4.x semaphore, if the semaphore is available

SYNOPSIS

```

STATUS semClear
(
  SEM_ID semId /* semaphore ID to empty */
)

```

DESCRIPTION

This routine takes a VxWorks 4.x semaphore if it is available (full), otherwise no action is taken except to return ERROR. This routine never preempts the caller.

RETURNS

OK, or ERROR if the semaphore is unavailable.

ERRNO

N/A

SEE ALSO

semOLib

2.8 **semBLib**

NAME

semBLib - binary semaphore library

ROUTINES

semBCreate() - create and initialize a binary semaphore

DESCRIPTION

This library provides the interface to VxWorks binary semaphores. Binary semaphores are the most versatile, efficient, and conceptually simple type of semaphore. They can be used to: (1) control mutually exclusive access to shared devices or data structures, or (2) synchronize multiple tasks, or task-level and interrupt-level processes. Binary semaphores form the foundation of numerous VxWorks facilities.

A binary semaphore can be viewed as a cell in memory whose contents are in one of two states, full or empty. When a task takes a binary semaphore, using semTake(), subsequent action depends on the state of the semaphore:

(1) If the semaphore is full, the semaphore is made empty, and the calling task continues executing.

(2) If the semaphore is empty, the task will be blocked, pending the availability of the semaphore. If a timeout is specified and the timeout expires, the pended task will be removed from the queue of pended tasks and enter the ready state with an ERROR status. A pended task is ineligible for CPU allocation. Any number of tasks may be pended simultaneously on the same binary semaphore.

When a task gives a binary semaphore, using semGive(), the next available task in the pend queue is unblocked. If no task is pending on this semaphore, the semaphore becomes full. Note that if a semaphore is given, and a task is unblocked that is of higher priority than the task that called semGive(), the unblocked task will preempt the calling task.

MUTUAL EXCLUSION

To use a binary semaphore as a means of mutual exclusion, first create it with an initial state of full. For example:

```
SEM_ID semMutex;
```

```
/* create a binary semaphore that is initially full */
semMutex = semBCreate (SEM_Q_PRIORITY, SEM_FULL);
```

Then guard a critical section or resource by taking the semaphore with semTake(), and exit the section or release the resource by giving the semaphore with semGive(). For example:

```
semTake (semMutex, WAIT_FOREVER);
... /* critical region, accessible only by one task at a time */
```

```
semGive (semMutex);
```

While there is no restriction on the same semaphore being given, taken, or flushed by multiple tasks, it is important to ensure the proper functionality of the mutual-exclusion construct. While there is no danger in any number of processes taking a semaphore, the giving of a semaphore should be more carefully controlled. If a semaphore is given by a task that did not take it, mutual exclusion could be lost.

SYNCHRONIZATION

To use a binary semaphore as a means of synchronization, create it with an initial state of empty. A task blocks by taking a semaphore at a synchronization point, and it remains blocked until the semaphore is given by another task or interrupt service routine.

Synchronization with interrupt service routines is a particularly common need. Binary semaphores can be given, but not taken, from interrupt level. Thus, a task can block at a synchronization point with semTake(), and an interrupt service routine can unblock that task with semGive().

In the following example, when init() is called, the binary semaphore is created, an interrupt service routine is attached to an event, and a task is spawned to process the event. Task 1 will run

until it calls `semTake()`, at which point it will block until an event causes the interrupt service routine to call `semGive()`. When the interrupt service routine completes, task 1 can execute to process the event.

```

SEM_ID semSync; /* ID of sync semaphore */

init()
{
intConnect (... , eventInterruptSvcRout, ...);
semSync = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
taskSpawn (... , task1);
}

task1()
{
...
semTake (semSync, WAIT_FOREVER); /* wait for event */
... /* process event */
}

eventInterruptSvcRout()
{
...
semGive (semSync); /* let task 1 process event */
...
}

```

A `semFlush()` on a binary semaphore will atomically unblock all pended tasks in the semaphore queue, i.e., all tasks will be unblocked at once, before any actually execute.

CAVEATS

There is no mechanism to give back or reclaim semaphores automatically when tasks are suspended or deleted. Such a mechanism, though desirable, is not currently feasible. Without explicit knowledge of the state of the guarded resource or region, reckless automatic reclamation of a semaphore could leave the resource in a partial state. Thus, if a task ceases execution unexpectedly, as with a bus error, currently owned semaphores will not be given back, effectively leaving a resource permanently unavailable. The mutual-exclusion semaphores provided by `semMLib` offer protection from unexpected task deletion.

INCLUDE FILES

`semLib.h`

SEE ALSO

`semLib`, `semCLib`, `semMLib`, VxWorks Programmer's Guide: Basic OS

API Reference: Routines

2.9 `semBCreate()`

NAME

`semBCreate()` - create and initialize a binary semaphore

SYNOPSIS

`SEM_ID semBCreate`

```

(
int    options,    /* semaphore options */
SEM_B_STATE initialState /* initial semaphore state */
)
```

DESCRIPTION

This routine allocates and initializes a binary semaphore. The semaphore is initialized to the `initialState` of either `SEM_FULL` (1) or `SEM_EMPTY` (0).

The options parameter specifies the queuing style blocked tasks and response on signals for blocked RTP tasks. Tasks may be queued on a priority basis or a first-in-first-out basis. The queuing style options are SEM_Q_PRIORITY (0x1) and SEM_Q_FIFO (0x0), respectively. That parameter also specifies if `semGive()` should return ERROR when the semaphore fails to send events. This option is turned off by default; it is activated by doing a bitwise-OR of SEM_EVENTSEND_ERR_NOTIFY (0x10) with the queuing style of the semaphore. SEM_INTERRUPTIBLE(0x20) is the option which makes the blocked RTP task on the semaphore ready and return ERROR with errno set to EINTR when a signal is generated to that task. This option has no affect when a kernel task blocks on the same semaphore created with this option. This option is turned off by default.

RETURNS

The semaphore ID, or NULL if memory cannot be allocated.

ERRNO

N/A

SEE ALSO

`semBLib`
`semCLib`

NAME

`semCLib` - counting semaphore library

ROUTINES

`semCCreate()` - create and initialize a counting semaphore

DESCRIPTION

This library provides the interface to VxWorks counting semaphores. Counting semaphores are useful for guarding multiple instances of a resource.

A counting semaphore may be viewed as a cell in memory whose contents keep track of a count. When a task takes a counting semaphore using `semTake()`, subsequent action depends on the state of the count:

- (1) If the count is non-zero, it is decremented and the calling task continues executing.
- (2) If the count is zero, the task is blocked, pending the availability of the semaphore. If a timeout is specified and the timeout expires, the pended task is removed from the queue of pended tasks and enters the ready state with an ERROR status. A pended task is ineligible for CPU allocation. Any number of tasks may be pended simultaneously on the same counting semaphore.

When a task gives a semaphore, using `semGive()`, the next available task in the pend queue is unblocked. If no task is pending on this semaphore, the semaphore count is incremented. Note that if a semaphore is given, and a task is unblocked that is of higher priority than the task that called `semGive()`, the unblocked task preempts the calling task.

A `semFlush()` on a counting semaphore atomically unblocks all pended tasks in the semaphore queue. This means all tasks are made ready before any task actually executes. The count of the semaphore remains unchanged.

INTERRUPT USAGE

Counting semaphores may be given but not taken from interrupt level.

CAVEATS

There is no mechanism to give back or reclaim semaphores automatically when tasks are suspended or deleted. Such a mechanism, though desirable, is not currently feasible. Without explicit knowledge of the state of the guarded resource or region, reckless automatic reclamation of a semaphore could leave the resource in a partial state. Thus, if a task ceases execution unexpectedly, as with a bus error, currently owned semaphores are not given back, effectively leaving a resource permanently unavailable. The mutual-exclusion semaphores provided by `semMLib` offer protection from unexpected task deletion.

INCLUDE FILES

`semLib.h`

SEE ALSO

semLib, semBLib, semMLib
API Reference: Routines

2.10 semCCreate()**NAME**

semCCreate() - create and initialize a counting semaphore

SYNOPSIS

```
SEM_ID semCCreate
(
    int options, /* semaphore option modes */
    int initialCount /* initial count */
)
```

DESCRIPTION

This routine allocates and initializes a counting semaphore. The semaphore is initialized to the specified initial count.

The options parameter specifies the queuing style and response on signals for blocked RTP tasks. Tasks may be queued on a priority basis or a first-in-first-out basis. The queuing style options are SEM_Q_PRIORITY (0x1) and SEM_Q_FIFO (0x0), respectively. That parameter also specifies if semGive() should return ERROR when the semaphore fails to send events. This option is turned off by default; it is activated by doing a bitwise-OR of SEM_EVENTSEND_ERR_NOTIFY (0x10) with the queuing style of the semaphore. SEM_INTERRUPTIBLE(0x20) is the option which makes the blocked RTP task on the semaphore ready and return ERROR with errno set to EINTR when a signal is generated to that task. This option has no affect when a kernel task blocks on the same semaphore created with this option. This option is turned off by default.

RETURNS

The semaphore ID, or NULL if memory cannot be allocated.

ERRNO

S_semLib_INVALID_INITIAL_COUNT

The specified initial count is negative

S_semLib_INVALID_OPTION

Options not applicable to counting semaphores were specified.

S_memLib_NOT_ENOUGH_MEMORY

There is not enough memory to create the semaphore.

SEE ALSO

semCLib

2.11 semMLib**NAME**

semMLib - mutual-exclusion semaphore library

ROUTINES

semMCreate() - create and initialize a mutual-exclusion semaphore

semMGiveForce() - give a mutual-exclusion semaphore without restrictions

DESCRIPTION

This library provides the interface to VxWorks mutual-exclusion semaphores. Mutual-exclusion semaphores offer convenient options suited for situations requiring mutually exclusive access to resources. Typical applications include sharing devices and protecting data structures. Mutual-exclusion semaphores are used by many higher-level VxWorks facilities.

The mutual-exclusion semaphore is a specialized version of the binary semaphore, designed to address issues inherent in mutual exclusion, such as recursive access to resources, priority inversion, and deletion safety. The fundamental behavior of the mutual-exclusion semaphore is

identical to the binary semaphore (see the manual entry for semBLib), except for the following restrictions:

- It can only be used for mutual exclusion.
- It can only be given by the task that took it.
- It may not be taken or given from interrupt level.
- The semFlush() operation is illegal.

These last two operations have no meaning in mutual-exclusion situations.

RECURSIVE RESOURCE ACCESS

A special feature of the mutual-exclusion semaphore is that it may be taken "recursively," i.e., it can be taken more than once by the task that owns it before finally being released. Recursion is useful for a set of routines that need mutually exclusive access to a resource, but may need to call each other.

Recursion is possible because the system keeps track of which task currently owns a mutual-exclusion semaphore. Before being released, a mutual-exclusion semaphore taken recursively must be given the same number of times it has been taken; this is tracked by means of a count which is incremented with each semTake() and decremented with each semGive().

The example below illustrates recursive use of a mutual-exclusion semaphore. Function A requires access to a resource which it acquires by taking semM; function A may also need to call function B, which also requires semM:

```
SEM_ID semM;
semM = semMCreate (...);
funcA ()
{
    semTake (semM, WAIT_FOREVER);
    ...
    funcB ();
    ...
    semGive (semM);
}

funcB ()
{
    semTake (semM, WAIT_FOREVER);
    ...
    semGive (semM);
}
```

PRIORITY-INVERSION SAFETY

If the option SEM_INVERSION_SAFE is selected, the library adopts a priority-inheritance protocol to resolve potential occurrences of "priority inversion," a problem stemming from the use of semaphores for mutual exclusion. Priority inversion arises when a higher-priority task is forced to wait an indefinite period of time for the completion of a lower-priority task.

Consider the following scenario: T1, T2, and T3 are tasks of high, medium, and low priority, respectively. T3 has acquired some resource by taking its associated semaphore. When T1 preempts T3 and contends for the resource by taking the same semaphore, it becomes blocked. If we could be assured that T1 would be blocked no longer than the time it normally takes T3 to finish with the resource, the situation would not be problematic. However, the low-priority task is vulnerable to preemption by medium-priority tasks; a preempting task, T2, could inhibit T3 from relinquishing the resource. This condition could persist, blocking T1 for an indefinite period of time.

The priority-inheritance protocol solves the problem of priority inversion by elevating the priority of T3 to the priority of T1 during the time T1 is blocked on T3. This protects T3, and indirectly T1, from preemption by T2. Stated more generally, the priority-inheritance protocol assures that a task which owns a resource will execute at the priority of the highest priority task blocked on that resource. Once the task priority has been elevated, it remains at the higher level until all

contributing mutual-exclusion semaphores that the task owns are released; then the task returns to its normal, or standard, priority. Hence, the "inheriting" task is protected from preemption by any intermediate-priority tasks.

The priority-inheritance protocol also takes into consideration a task's ownership of more than one mutual-exclusion semaphore at a time. Such a task will execute at the priority of the highest priority task blocked on any of its owned resources. Under most circumstances, the task will return to its normal priority only after relinquishing all contributing mutual-exclusion semaphores.

The sole exception to this occurs if some task tried to lower the priority of a task involved in priority inheritance by using taskPrioritySet(). This act creates some uncertainties in that task's inheritance tracking that can only be made certain when all the inversion safe mutual-exclusion semaphores that task has are known to be involved in its priority inheritance. If all previously known contributing mutual-exclusion semaphores were to be relinquished, the priority of that task might not be lowered to its newly assigned lower priority. It will however at least be lowered to the last known "safe" value-- typically the task's normal priority before that call to taskPrioritySet(). The priority can not be restored to the new normal priority before these uncertainties are removed. At the absolute worst, this is not until the task gives up all of its inversion safe mutual-exclusion semaphores.

SEMAPHORE DELETION

The semDelete() call terminates a semaphore and deallocates any associated memory. The deletion of a semaphore unblocks tasks pended on that semaphore; the routines which were pended return ERROR. Take special care when deleting mutual-exclusion semaphores to avoid deleting a semaphore out from under a task that already owns (has taken) that semaphore. Applications should adopt the protocol of only deleting semaphores that the deleting task owns.

TASK-DELETION SAFETY

If the option SEM_DELETE_SAFE is selected, the task owning the semaphore will be protected from deletion as long as it owns the semaphore. This solves another problem endemic to mutual exclusion. Deleting a task executing in a critical region can be catastrophic. The resource could be left in a corrupted state and the semaphore guarding the resource would be unavailable, effectively shutting off all access to the resource.

As discussed in taskLib, the primitives taskSafe() and taskUnsafe() offer one solution, but as this type of protection goes hand in hand with mutual exclusion, the mutual-exclusion semaphore provides the option SEM_DELETE_SAFE, which enables an implicit taskSafe() with each semTake(), and a taskUnsafe() with each semGive(). This convenience is also more efficient, as the resulting code requires fewer entrances to the kernel.

CAVEATS

There is no mechanism to give back or reclaim semaphores automatically when tasks are suspended or deleted. Such a mechanism, though desirable, is not currently feasible. Without explicit knowledge of the state of the guarded resource or region, reckless automatic reclamation of a semaphore could leave the resource in a partial state. Thus if a task ceases execution unexpectedly, as with a bus error, currently owned semaphores will not be given back, effectively leaving a resource permanently unavailable. The SEM_DELETE_SAFE option partially protects an application, to the extent that unexpected deletions will be deferred until the resource is released.

Because the priority of a task which has been elevated by the taking of a mutual-exclusion semaphore remains at the higher priority until all mutexes held by that task are released, unbounded priority inversion situations can result when nested mutexes are involved. If nested mutexes are required, consider the following alternatives:

1. Avoid overlapping critical regions.
2. Adjust priorities of tasks so that there are no tasks at intermediate priority levels.
3. Adjust priorities of tasks so that priority inheritance protocol is not needed.
4. Manually implement a static priority ceiling protocol using a non-inversion-save mutex. This involves setting all blockers on a mutex to the ceiling priority, then taking the mutex. After semGive, set the priorities back to the base priority. Note that this implementation reduces the queue to a fifo queue.

INCLUDE FILES

semLib.h

SEE ALSOsemLib, semBLib, semCLib, VxWorks Programmer's Guide: Basic OS
API Reference: Routines**2.12 semMCreate()****NAME**

semMCreate() - create and initialize a mutual-exclusion semaphore

SYNOPSIS

```
SEM_ID semMCreate
(
    int options /* mutex semaphore options */
)
```

DESCRIPTION

This routine allocates and initializes a mutual-exclusion semaphore. The semaphore state is initialized to full.

Semaphore options include the following:

SEM_Q_PRIORITY (0x1)

Queue pended tasks on the basis of their priority.

SEM_Q_FIFO (0x0)

Queue pended tasks on a first-in-first-out basis.

SEM_DELETE_SAFE (0x4)

Protect a task that owns the semaphore from unexpected deletion. This option enables an implicit taskSafe() for each semTake(), and an implicit taskUnsafe() for each semGive().

SEM_INVERSION_SAFE (0x8)

Protect the system from priority inversion. With this option, the task owning the semaphore will execute at the highest priority of the tasks pended on the semaphore, if it is higher than its current priority. This option must be accompanied by the SEM_Q_PRIORITY queuing mode.

SEM_EVENTSEND_ERR_NOTIFY (0x10)

When the semaphore is given, if a task is registered for events and the actual sending of events fails, a value of ERROR is returned and the errno is set accordingly. This option is off by default.

SEM_INTERRUPTIBLE (0x20)

Signal sent to an RTP task blocked on a semaphore created with this option, would make the task ready and return with ERROR and errno set to EINTR. This option has no affect for a kernel task blocked on the same semaphore created with this option. This option is off by default.

RETURNS

The semaphore ID, or NULL if the semaphore cannot be created.

ERRNO

S_semLib_INVALID_OPTION

Invalid option was passed to semMCreate.

S_memLib_NOT_ENOUGH_MEMORY

Not enough memory available to create the semaphore.

SEE ALSOsemMLib, semLib, semBLib, taskSafe(), taskUnsafe()
API Reference: Routines

2.13 **semMGiveForce()**

NAME

semMGiveForce() - give a mutual-exclusion semaphore without restrictions

SYNOPSIS

```
STATUS semMGiveForce
(
  FAST SEM_ID semId /* semaphore ID to give */
)
```

DESCRIPTION

This routine gives a mutual-exclusion semaphore, regardless of semaphore ownership. It is intended as a debugging aid only.

The routine is particularly useful when a task dies while holding some mutual-exclusion semaphore, because the semaphore can be resurrected. The routine will give the semaphore to the next task in the pend queue or make the semaphore full if no tasks are pending. In effect, execution will continue as if the task owning the semaphore had actually given the semaphore.

CAVEATS

This routine should be used only as a debugging aid, when the condition of the semaphore is known.

RETURNS

OK, or ERROR if the semaphore ID is invalid.

ERRNO

N/A

SEE ALSO

semMLib, semGive()

3 Gestion de boîtes aux lettres

3.1 **msgQLib**

NAME

msgQLib - message queue library

ROUTINES

- msgQCreate() - create and initialize a message queue
- msgQDelete() - delete a message queue
- msgQSend() - send a message to a message queue
- msgQReceive() - receive a message from a message queue
- msgQNumMsgs() - get the number of messages queued to a message queue

DESCRIPTION

This library contains routines for creating and using message queues, the primary intertask communication mechanism within a single CPU. Message queues allow a variable number of messages (varying in length) to be queued in first-in-first-out (FIFO) order. Any task or interrupt service routine can send messages to a message queue. Any task can receive messages from a message queue. Multiple tasks can send to and receive from the same message queue. Full-duplex communication between two tasks generally requires two message queues, one for each direction.

To provide message queue support for a system, VxWorks must be configured with the INCLUDE_MSG_Q component.

CREATING AND USING MESSAGE QUEUES

A message queue is created with msgQCreate(). Its parameters specify the maximum number of messages that can be queued to that message queue and the maximum length in bytes of each message. Enough buffer space is pre-allocated to accommodate the specified number of messages of the specified length.

A task or interrupt service routine sends a message to a message queue with `msgQSend()`. If no tasks are waiting for messages on the message queue, the message is added to the buffer of messages for that queue. If any tasks are already waiting to receive a message from the message queue, the message is immediately delivered to the first waiting task.

A task receives a message from a message queue with `msgQReceive()`. If any messages are already available in the message queue's buffer, the first message is immediately dequeued and returned to the caller. If no messages are available, the calling task blocks and is added to a queue of tasks waiting for messages. This queue of waiting tasks can be ordered either by task priority or FIFO, as specified in an option parameter when the queue is created.

TIMEOUTS

Both `msgQSend()` and `msgQReceive()` take timeout parameters. When sending a message, if no buffer space is available to queue the message, the timeout specifies how many ticks to wait for space to become available. When receiving a message, the timeout specifies how many ticks to wait if no message is immediately available. The timeout parameter can have the special values `NO_WAIT` (0) or `WAIT_FOREVER` (-1). `NO_WAIT` means the routine returns immediately; `WAIT_FOREVER` means the routine never times out.

URGENT MESSAGES

The `msgQSend()` routine allows the priority of a message to be specified. It can be either `MSG_PRI_NORMAL` (0) or `MSG_PRI_URGENT` (1). Normal priority messages are added to the tail of the list of queued messages, while urgent priority messages are added to the head of the list.

VXWORKS EVENTS

If a task has registered with a message queue via `msgQEvnStart()`, events are sent to that task when a message arrives on that message queue, on the condition that no other task is pending on the queue.

CAVEATS

There is a small difference between how pended senders and pended receivers are handled. As in previous versions of VxWorks, a pended receiver is made ready as soon as a sender sends a message.

Unlike previous versions of VxWorks, FIFO message queues allow only one pended sender can be made ready by a receive; subsequent receive operations do not unpend more senders. Instead the next sender is unpended by the previously unpended sender. This enforces the correct order of delivery of messages onto the queue. This may affect the length of time a sender spends pending for the message queue resource. Priority message queues are not affected by this restriction.

INCLUDE FILES

`msgQLib.h`

SEE ALSO

`pipeDrv`, `msgQSmLib`, `msgQEvnLib`, `eventLib`

API Reference: Routines

3.2 `msgQCreate()`

NAME

`msgQCreate()` - create and initialize a message queue

SYNOPSIS

```
MSG_Q_ID msgQCreate
(
    int maxMsgs,    /* max messages that can be queued */
    int maxMsgLength, /* max bytes in a message */
    int options    /* message queue options */
)
```

DESCRIPTION

This routine creates a message queue capable of holding up to maxMsgs messages, each up to maxMsgLength bytes long. The routine returns a message queue ID used to identify the created message queue in all subsequent calls to routines in this library. The queue can be created with the following options:

MSG_Q_FIFO (0x00)

Queue pended tasks in FIFO order.

MSG_Q_PRIORITY (0x01)

Queue pended tasks in priority order.

MSG_Q_EVENTSEND_ERR_NOTIFY (0x02)

When a message is sent, if a task is registered for events and the actual sending of events fails, a value of ERROR is returned and errno is set. This option is off by default.

MSG_Q_INTERRUPTIBLE (0x04)

Signal sent to a RTP task pended on a message queue created with this option, would make the task ready and return ERROR with errno set to EINTR. This option has no affect for a kernel task pended on the same message queue created with this option. This option is off by default.

RETURNS

The MSG_Q_ID, or NULL if error.

ERRNO

S_memLib_NOT_ENOUGH_MEMORY

There is not enough memory to create the queue as specified.

S_intLib_NOT_ISR_CALLABLE

This routine cannot be called from interrupt level.

S_msgQLib_INVALID_MSG_LENGTH

Negative maxMsgLength specified.

S_msgQLib_INVALID_MSG_COUNT

Negative maxMsgs specified.

S_msgQLib_INVALID_QUEUE_TYPE

Invalid pending queue type specified.

S_msgQLib_ILLEGAL_OPTIONS

Illegal option bits were specified.

SEE ALSO

msgQLib, msgQSmLib

API Reference: Routines

3.3 *msgQDelete()*

NAME

msgQDelete() - delete a message queue

SYNOPSIS

STATUS msgQDelete

(

MSG_Q_ID msgQId /* message queue to delete */

)

DESCRIPTION

This routine deletes a message queue. All tasks pending on either msgQSend() or msgQReceive(), or pending for the reception of events meant to be sent from the message queue, unblock and return ERROR. When this function returns, msgQId is no longer a valid message queue ID.

RETURNS

OK on success or ERROR otherwise.

ERRNO

S_objLib_OBJ_ID_ERROR

The message queue ID is invalid.

S_intLib_NOT_ISR_CALLABLE

The routine cannot be called from interrupt level.

S_distLib_NO_OBJECT_DESTROY

Deleting a distributed message queue is not permitted.

S_smObjLib_NO_OBJECT_DESTROY

Deleting a shared message queue is not permitted.

S_objLib_OBJ_OPERATION_UNSUPPORTED

Deleting a named message queue is not permitted.

SEE ALSO

msgQLib, msgQSmLib

API Reference: Routines

3.4 msgQSend()

NAME

msgQSend() - send a message to a message queue

SYNOPSIS

STATUS msgQSend

```
(  
    MSG_Q_ID msgQId, /* message queue on which to send */  
    char * buffer, /* message to send */  
    UINT nBytes, /* length of message */  
    int timeout, /* ticks to wait */  
    int priority /* MSG_PRI_NORMAL or MSG_PRI_URGENT */  
)
```

DESCRIPTION

This routine sends the message in buffer of length nBytes to the message queue msgQId. If any tasks are already waiting to receive messages on the queue, the message is immediately delivered to the first waiting task. If no task is waiting to receive messages, the message is saved in the message queue and, if a task has previously registered to receive events from the message queue, these events are sent in the context of this call. This may result in the unpending of the task waiting for the events. If the message queue fails to send events, and if it was created using the MSG_Q_EVENTSEND_ERR_NOTIFY option, ERROR is returned even though the message was successfully sent to the queue.

The timeout parameter specifies the number of ticks to wait for adding its message to the queue if the message queue is full. The timeout parameter can also have the following special values:

NO_WAIT (0)

Return immediately, even if the message has not been sent.

WAIT_FOREVER (-1)

Never time out.

The priority parameter specifies the priority of the message being sent. The possible values are:

MSG_PRI_NORMAL (0)

Normal priority; add the message to the tail of the list of queued messages.

MSG_PRI_URGENT (1)

Urgent priority; add the message to the head of the list of queued messages.

USE BY INTERRUPT SERVICE ROUTINES

This routine can be called by interrupt service routines as well as by tasks. This is one of the primary means of communication between an ISR and a task. When called from an ISR, timeout must be NO_WAIT.

RETURNS

OK on success or ERROR otherwise.

ERRNO

S_distLib_NOT_INITIALIZED

The distributed objects message queue library (VxFusion) was not initialized.

S_smObjLib_NOT_INITIALIZED

The shared memory message queue library (VxMP Option) was not initialized.
S_objLib_OBJ_ID_ERROR
 The message queue ID is invalid.
S_objLib_OBJ_DELETED
 The message queue was deleted while the calling task was pended.
S_objLib_OBJ_UNAVAILABLE
 No free buffer space was available and the NO_WAIT timeout was specified.
S_objLib_OBJ_TIMEOUT
 A timeout occurred while waiting for buffer space to become available.
S_msgQLib_INVALID_MSG_LENGTH
 The message length exceeds the limit.
S_msgQLib_NON_ZERO_TIMEOUT_AT_INT_LEVEL
 The routine was called from an ISR with a non-zero timeout.
S_eventLib_EVENTSEND_FAILED
 The message queue failed to send events to the registered task. This errno value can occur only if the message queue was created with the MSG_Q_EVENTSEND_ERR_NOTIFY option.

SEE ALSO

msgQLib, msgQSmLib, msgQEvStart()
 API Reference: Routines

3.5 *msgQReceive()*

NAME

msgQReceive() - receive a message from a message queue

SYNOPSIS

```
int msgQReceive
(
  MSG_Q_ID msgQId, /* message queue from which to receive */
  char * buffer, /* buffer to receive message */
  UINT maxNBytes, /* length of buffer */
  int timeout /* ticks to wait */
)
```

DESCRIPTION

This routine receives a message from the message queue msgQId. The received message is copied into the specified buffer, which is maxNBytes in length. If the message is longer than maxNBytes, the remainder of the message is discarded (no error indication is returned).

The timeout parameter specifies the number of ticks to wait for a message to be sent to the queue, if no message is available when msgQReceive() is called. The timeout parameter can also have the following special values:

NO_WAIT (0)

Return immediately, whether a message has been received or not.

WAIT_FOREVER (-1)

Never time out.

WARNING

This routine must not be called by interrupt service routines.

RETURNS

The number of bytes copied to buffer, or ERROR.

ERRNO

S_distLib_NOT_INITIALIZED

The distributed objects message queue library (VxFusion) was not initialized.

S_smObjLib_NOT_INITIALIZED

The shared memory message queue library (VxMP Option) was not initialized.

S_objLib_OBJ_ID_ERROR

The message queue ID is invalid.
S_objLib_OBJ_DELETED
 The message queue was deleted while the calling task was pended.
S_objLib_OBJ_UNAVAILABLE
 No message was available and the NO_WAIT timeout was specified.
S_objLib_OBJ_TIMEOUT
 A timeout occurred while waiting for a message to become available.
S_msgQLib_INVALID_MSG_LENGTH
 The message length exceeds the limit.
S_intLib_NOT_ISR_CALLABLE
 This routine cannot be called from interrupt level.
SEE ALSO
 msgQLib, msgQSmLib
 API Reference: Routines

3.6 **msgQNumMsgs()**

NAME

msgQNumMsgs() - get the number of messages queued to a message queue

SYNOPSIS

```
int msgQNumMsgs
(
  FAST MSG_Q_ID msgQId /* message queue to examine */
)
```

DESCRIPTION

This routine returns the number of messages currently queued to a specified message queue.

RETURNS

The number of messages queued, or ERROR.

ERRNO

S_distLib_NOT_INITIALIZED
 The distributed objects message queue library (VxFusion) was not initialized.
S_smObjLib_NOT_INITIALIZED
 The shared memory message queue library (VxMP Option) was not initialized.
S_objLib_OBJ_ID_ERROR
 The message queue ID is invalid.

SEE ALSO

msgQLib, msgQSmLib

4 Liste des primitives ou bibliothèques

1	Gestion des tâches	Doc_VxW - 1
1.1	taskLib	Doc_VxW - 1
1.2	taskActivate()	Doc_VxW - 3
1.3	taskDelay()	Doc_VxW - 3
1.4	taskDelete()	Doc_VxW - 4
1.5	taskIdSelf()	Doc_VxW - 4
1.6	taskInit()	Doc_VxW - 6
1.7	taskLock()	Doc_VxW - 7
1.8	taskPriorityGet()	Doc_VxW - 7
1.9	taskPrioritySet()	Doc_VxW - 8
1.10	taskResume()	Doc_VxW - 8
1.11	taskSpawn()	Doc_VxW - 9
1.12	taskSuspend()	Doc_VxW - 10
2	Gestion des semaphores	Doc_VxW - 11
2.1	semLib	Doc_VxW - 11
2.2	semGive()	Doc_VxW - 12
2.3	semTake()	Doc_VxW - 13
2.4	semFlush()	Doc_VxW - 14
2.5	semDelete()	Doc_VxW - 14
2.6	semInit()	Doc_VxW - 15
2.7	semClear()	Doc_VxW - 15
2.8	semBLib	Doc_VxW - 16
2.9	semBCreate()	Doc_VxW - 17
2.10	semCCreate()	Doc_VxW - 19
2.11	semMLib	Doc_VxW - 19
2.12	semMCreate()	Doc_VxW - 22
2.13	semMGiveForce()	Doc_VxW - 23
3	Gestion de boîtes aux lettres	Doc_VxW - 23
3.1	msgQLib	Doc_VxW - 23
3.2	msgQCreate()	Doc_VxW - 24
3.3	msgQDelete()	Doc_VxW - 25
3.4	msgQSend()	Doc_VxW - 26
3.5	msgQReceive()	Doc_VxW - 27
3.6	msgQNumMsgs()	Doc_VxW - 28