Hes·so // VALAIS WALLIS
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences Western Switzerland

Domaine Sciences de l'ingénieur
Rte du Rawyl 47
CH- 1950 Sion 2
Tél. +41 27 606 85 11
Fax +41 27 606 85 75
info@hevs.ch

www.hevs.ch

Filière Systèmes industriels

Orientation Infotronics

Diplôme 2007

Joël Rochat

Environnement de cross-développement libre pour ARM EBS3

Professeur	Pierre Pompili
Expert	Daniel Rossier

Sion, le 23 novembre 2007

HES-SO Valais

SI	TV	EE	IG	EST
X	X	X	X	

Données du travail de diplôme
Daten der Diplomarbeit

FO.0.2.02.07.DB
DD/18/05/2006

Filière / *Studiengang* : Systèmes industriels

Confidentiel / *Vertraulich* ☐

Etudiant / *Student* Joël Rochat	**Année scolaire / *Schuljahr*** 2006/07	**No TD / *Nr. DA*** SI/2007/10
Proposé par / *vorgeschlagen von* HES-SO Valais, UIT		**Lieu d'exécution / *Ausführungsort*** HES-SO Valais, DSI **Expert / *Experte***

Titre / *Titel*:

Environnement de cross-développement libre pour ARM EBS3

Description / *Beschreibung*:

La qualité des outils de développement issus du monde libre ne cesse de s'améliorer. Dans le but de remplacer à terme l'environnement de cross-développement MULTI 2000, le laboratoire de systèmes embarqués désire mettre en œuvre plusieurs outils du domaine public.

Ce projet consiste à bâtir, autour d'Eclipse, une plateforme ouverte de cross-développement pour déployer et contrôler les logiciels embarqués sur cibles ARM. Cette plateforme devra prendre en compte tout le cycle de vie d'un programme :

- spécification formelle
- édition et compilation
- mise au point du programme
- analyse temporelle
- gestion de la documentation
- gestion des versions.

Objectifs / *Ziele:*

— Mettre en place des composants logiciels pour la prise en compte du cycle de vie d'un programme
— Paramétrer l'environnent Eclipse pour cross-développer l'ARM EBS3 de manière conviviale
— Porter RTEMS (Real-Time Operating System for Multiprocessor Systems) sur l'ARM EBS3
— Documenter l'ensemble des paramètres modifiés dans Eclipse pour arriver aux objectifs fixés

Signature ou visa / *Unterschrift oder Visum*

Resp. de l'orientation infotronics
Pompili R.
.....

Professeur/*Dozent*: Pierre Pompili
Pompili R.
.....

Etudiant/*Student*: Rochat.
.....

Délais / *Termine*

Attribution du thème / Ausgabe des Auftrags:
03.09.2007

Remise du rapport / Abgabe des Schlussberichts:
23.11.2007

Exposition publique / Ausstellung Diplomarbeiten:
30.11.2007

Défenses orales / Mündliche Verfechtungen
Semaine 49

Rapport reçu le / *Schlussbericht erhalten am* ..... Visa du secrétariat / *Visum des Sekretariats:* .....

Joël Rochat 2007

Environnement de cross-développement libre pour ARM EBS3

Freie croß-developpement Umgebung für ARM EBS3

Objectif

Mettre en place des composants logiciels pour la prise en compte du cycle de vie d'un programme. Paramétrer l'environnement Eclipse pour cross-développer l'ARM EBS3. Porter RTEMS sur l'ARM EBS3. Documenter le développement du plugin Eclipse

Résultats

Le système réalisé permet de développer des programmes standalones ou multitâches entièrement depuis Eclipse. La gestion de la communication entre le PC et la cible est entièrement gérée. Les outils mis à disposition par *RTEMS* fonctionnent correctement sur l'ARM EBS3

Mots-clés

cross development, RTEMS, Eclipse, plugin, GNU

Ziel

Ein Eclipse Plugin für das Cross-Development des ARM EBS3 zu verwirklichen. Das RTEMS Betriebssystem auf ARM EBS3 zu portieren.

Resultate

Mit diesem Plugin kann man Standalone- oder Multitasking-Programme entwickeln. Die Kommunikation zwischen dem PC und ARM EBS3 ist automatisiert. Die RTEMS Tools wurden auf ARM EBS3 umgesetzt

Schlüsselwörter

cross development, RTEMS, Eclipse, plugin, GNU

Table des matières

Introduction

Afin de valider les trois années d'études passées à la HES SO de Sion un travail de diplôme est demandé à chaque étudiant. Ce projet individuel dure douze semaines et se conclut par la remise d'un rapport ainsi qu'une présentation d'une vingtaine de minutes. Le rapport doit permettre de comprendre, de reproduire et de poursuivre le travail effectué. Un projet se déroulant le dernier semestre de troisième année a permis de clarifier et d'organiser le travail à effectuer pour l'obtention du diplôme.

But

Le but de mon projet est de développer un plugin sur la plate-forme Eclipse qui permettra de programmer et de debugger la carte utilisée lors des laboratoires de 3ème année : l'ARMEBS3.

Le plugin devra être le plus « user frendly » pour que les utilisateurs perdent le moins de temps possible à configurer les chemins vers les utilitaires et les différents fichiers utilisés lors du développement d'un programme. Le principal attrait de ce travail de diplôme est le fait que tous les outils utilisés seront libres. A terme il sera envisageable de se passer de l'environnement utilisé jusqu'à ce jour : *MULTI 2000*.

La deuxième partie consiste à porter un système d'exploitation temps réel libre sur cette même carte. Au final, les utilisateurs pourront développer des logiciels *standalones* ou *multitâches*.

Cahier des charges

- L'utilisateur final aura la possibilité de créer un nouveau projet simplement en utilisant le wizare d'Eclipse. Par projet on entend un exécutable, une librairie ou un programme sur un OS temps réel
- La configuration des projets se fera lors de sa création ou à l'aide d'une fenêtre regroupant la globalité des informations nécessaires.
- La configuration par défaut d'un projet devra inclure les fichiers et informations standards pour pouvoir lancer l'exécution instantanément.
- L'interface de communication *OpenOCD* devra être gérée directement par l'IDE.
- Les outils mis à disposition par le système d'exploitation devront fonctionner correctement sur l'ARMEBS3

L'ARMEBS3 v1.3

La carte utilisée lors des laboratoires de systèmes embarqués et de programmation en temps réel est l'ARMEBS3. Cette carte est munie d'un processeur AT91RM9200 cadencé à 180MHz. Elle possède 128 Mbytes de mémoire SDRAM et 16Mbytes de Flash. Au niveau de sa connectique on retrouve deux ports USB, une connexion Ethernet, quatre boutons, deux leds ainsi que dix-sept entrées/sorties programmable (input, output, interrupt, ...). Pour communiquer avec l'ARMEBS3 (via un terminal) deux ports série (RS232) sont à disposition.

Outils utilisés

Cette section énumère et décrit les différents outils utilisés pour pouvoir compiler le code, développer le plugin et accéder à la cible. Comme expliqué dans le cahier des charges, ces outils sont libres.

Eclipse

L'IDE

Eclipse est un environnement de développement intégré (le terme *Eclipse* désigne également le projet correspondant, lancé par IBM) extensible, universel et polyvalent, permettant potentiellement de créer des projets de développement mettant en œuvre n'importe quel langage de programmation. Eclipse est écrit en Java, ce langage est également utilisé pour écrire des plugins.

La spécificité d'Eclipse vient du fait de son architecture totalement développée autour de la notion de plugins : toutes les fonctionnalités de cet atelier logiciel sont développées en tant que plugin. *Source : wikipedia.org*

CDT *(C/C++ Development Tooling)*

Le plugin CDT permet de « transformer » Eclipse en un IDE performant pour le développement d'applications C/C++. Ce plugin ajoute une nouvelle perspective (organisation des fenêtres) à Eclipse et permet de développer, de compiler et de debugger des applications C/C++. CDT utilise GCC ou *GNU Compiler Collection* qui est une collection de compilateurs très efficaces fournit par la communauté open source. Il permet de compiler du C (*gcc*) et du C++ (*g++*). Pour construire les applications développées ce plugin génère un *makefile* approprié via les informations collectées durant la création du projet et utilise ensuite la commande *make* pour générer l'exécutable.

PDE *(Plugin Development Environment)*

Eclipse met à disposition un outil permettant de simplifier le développement de plugins. Cet outil comporte une vue spécifique et met à disposition trois outils principaux à savoir :

- Un assistant mettant à disposition des modèles de plugins (du simple bouton à l'éditeur plus complet).
- Un éditeur du fichier *plugin.xml* regroupant toutes les informations relatives au développement du plugin.
- Un support permettant d'exécuter une deuxième instance d'Eclipse pour pouvoir tester et debugger le plugin en cours de développement.

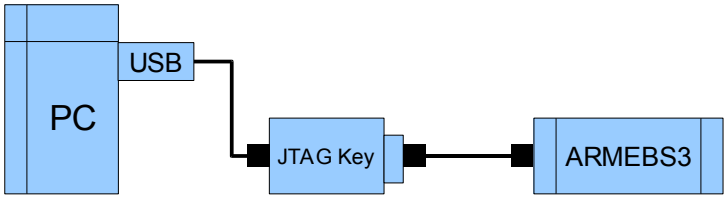
JTAG

Le **JTAG** pour **J**oint **T**est **A**ction **G**roup est le nom de la norme IEEE 1149.1 intitulé *"Standard Test Access Port and Boundary-Scan Architecture"*. La technique de Boundary-Scan (littéralement, *scrutation des frontières*) est conçue pour faciliter et automatiser le test des cartes électroniques numériques. La norme JTAG est aussi utilisée pour programmer et debugger les microprocesseurs en donnant l'accès direct à l'intérieur de celui-ci (lecture et écriture des registres,

accès aux mémoires et à la flash). La plupart des microprocesseurs y compris notre ARM920T intègre un bloc nommé *JTAG Scan* qui permet à la JTAG Key de la société Amontec d'accéder à l'ARMEBS3. *Source : wikipedia.org*

JTAG Key

La JTAG Key est un adaptateur permettant de relier l'interface JTAG de l'ARMEBS3 au port USB d'un PC. Cette clé fournie par la société Amontec permet de télécharger du code dans la cible, de debugger celui-ci et d'accéder aux registres du microprocesseur. De plus, des drivers Windows et Linux sont fournis et la prise enfichable est compatible avec l'entrée 20-pins du microprocesseur ARM.



OpenOCD (On-Chip Debugger)

OpenOCD est une interface de communication entre un PC et un système embarqué. C'est un outil libre permettant de programmer notamment des cibles basées sur ARM7 et ARM9 implémentant une interface JTAG. Il permet d'utiliser le debugger *gdb* compilé pour l'architecture ARM en envoyant des commandes spécifiques à travers un serveur telnet. *OpenOCD* transcrit les instructions *gdb* reçue en commandes JTAG. A noter qu'il est possible d'envoyer des commandes spécifiques uniquement à *OpenOCD* pour le configurer (enable break points, reset, ...).

YAGARTO

On entend par *toolchain* un ensemble d'outils permettant de compiler, lier et debugger un exécutable. YAGARTO met à disposition des outils spécifiques aux microprocesseurs ARM. Ces outils sont dérivés de GCC (GNU Compiler Collection) qui est une collection de compilateurs très efficaces fournit par la communauté open source. Cette *toolchain* inclut les compilateurs C et C++ (*arm-elf-gcc, arm-elf-g++)* ainsi que plusieurs autres utilitaires y compris le debugger (*arm-elf-gdb*). Le format de l'exécutable généré est *.elf*.

RTEMS

Real Time Executive for Multiprocessor Systems est un système d'exploitation temps réel libre. Destiné à la base pour des applications militaires, il est aujourd'hui utilisé pour développer des systèmes embarqués lorsque le temps de réponse et la réactivité sont des contraintes fortes (dites temps-réel dur). *RTEMS* supporte différents type de standard comme POSIX ou encore la pile TCP/IP développée à la base par le projet BSD. La société OAR Corporation gère actuellement le projet *RTEMS*. Ce système est actuellement utilisé dans beaucoup de domaines : communication, médical, espace et aviation, robotique, son et musique, etc. *Source : wikipedia.org*

RTEMS Toolchain

De nombreuses *toolchains* sont disponibles pour compiler *RTEMS* en rapport avec le processeur utilisé. Nous utiliserons donc la toolchain *arm-rtems4.7* sur linux et sur windows.

VMware Server

Cet outils permet de faire fonctionner plusieurs systèmes d'exploitations en même temps sur une seule et même machine physique grâce à la technologie de la virtualisation. *VMware* nous a permis de démarrer une instance de la distribution Linux *Fedora Core 5*. Après plusieurs essais infructueux sur Windows, nous avons décidé de passé vers le monde libre pour la compilation de l'exécutif *RTEMS*. *Fedora Core* (sponsorisé par *Red Hat*) permet de mieux gérer les outils nécessaires via le systèmes de paquets rpm.

Versions utilisées

- Eclipse : 3.3
- OpenOCD : r204
- Yagarto : binutils:2.17 gcc:4.2.1 newlib:1.15.0 insight:6.6
- RTEMS : 4.7.1
- VMWare Server : 1.0.4
- RTEMS toolchain : rtems4.7-arm-3

Debugger un *.elf* avec ces outils

Executable and Linking Format est un format de fichiers informatiques binaires utilisés pour l'enregistrement de code compilé. Ce format est utilisé dans la plupart des systèmes d'exploitations Unix. Chaque fichier *elf* est constitué d'un en-tête fixe, puis de segments et de sections. Les segments contiennent les informations nécessaires à l'exécution du programme contenu dans le fichier.

Les fichiers utilisés

crt.s (*c runtime*)

Le fichier *crt.s* regroupe un certain nombre de routines de démarrage du programme. Il déclare entre autre la table des vecteurs ainsi que la taille de la pile nécessaire à chaque interruption (reset, FIQ, IRQ...). Il est utilisé lors de la compilation du programme.

Il gère entre autre :

- la table des vecteurs et l'appel des sous-routines lors d'un reset ou d'une interruption.
- la copie des différentes sections de l'exécutable.
- l'appel de la fonction *main*.

Annexe 1.1 le fichier crt.s

armebs3_ram.ld

Le fichier utilisé par l'éditeur de liens porte en général l'extension *.ld*. Il est utilisé par le linker pour organiser et placer les différentes sections du programme aux bons endroit dans le fichier exécutable *.elf*. Un programme standard est divisé en au moins trois sections, à savoir :

- *.text* : contient le code du programme.
- *.data* : regroupe les variables globales du code.
- *.bss* : regroupe les variables non initialisées du programme.

De plus, ce fichier permet de définir des symboles qui seront ensuite utilisés par le programme de démarrage pour connaître l'emplacement mémoire des différentes sections du programme. Pour la compilation d'un projet *C standalone* un fichier de lien très simple a été utilisé. Celui-ci spécifie un certain nombre de chose : l'origine de la RAM ainsi que sa taille, les sections utilisées et leur emplacement dans la mémoire.

```
MEMORY
{
  ram : org = 0x20000000, len = 128M
}
```

Bloc « memory »

Spécification de l'origine de la RAM ainsi que sa longueur. Organisation de la section *.text* en spécifiant plusieurs informations notamment des « sous-sections », leur alignement sur 4 bytes etc... la dernière ligne indique que cette section est envoyée dans le premier emplacement libre de la RAM. Pour la première section cet emplacement se situe à l'adresse 0x20000000 (org).

Bloc « sections »

```
SECTIONS
{
  .text :
  {
    *(.vectors);
    . = ALIGN(4);
    *(.init);
    . = ALIGN(4);
    *(.text);
    . = ALIGN(4);
    *(.rodata);
    . = ALIGN(4);
    *(.rodata*);
    . = ALIGN(4);
    *(.glue_7t);
    . = ALIGN(4);
    *(.glue_7);
    . = ALIGN(4);
    etext = .;
  } > ram
```

Première section

Envoie dans la ram

Annexe 1.2 le fichier armebs3_ram.ld

openOCD.cfg

Pour utiliser *OpenOCD* correctement il faut le lancer en mode console en lui passant un fichier de configuration. Ce fichier contient différents flags permettant de le configurer pour la cible que nous utilisons. Deux exécutables sont à disposition, à savoir *openocd-pp.exe* qui est utilisé si la cible est connectée à l'ordinateur via le port parallèle et *openocd-ftd2xx* pour une connexion usb-JTAG. Lors du démarrage de ce logiciel un fichier de configuration est passé en paramètre. Ce fichier n'envoie aucune commande au processeur mais établit une connexion avec la cible.

Voici les principaux modules configurés pour utiliser *OpenOCD* avec la *JTAG Key*:

- telnet_port 4444, gdb_port 3333 : spécifie les ports sur lesquelles *OpenOCD* reçoit les connections *telnet* et *gdb*.
- interface ft2232 : permet de spécifier l'interface utilisée pour la JTAG Key.
- jtag_<option> : permet de configurer les options spécifiques à la JTAG Key. Par exemple jtag_speed pour la vitesse maximale de l'interface.
- jtag_device : décrit le dispositif utilisé
  - <IR length> : longueur du registre d'instruction
  - <IR capture> : valeur capturée durant la Capture-IR
  - <IR mask> : masque utilisé pour valider une instruction exécutée
  - <IDCODE> : instruction spécifique permettant de récupérer le code d'identification JTAG du device.

Exemple : jtag_device 4 0x1 0xf 0xe

- target : décrit la cible qui va être debuggée
  - <type> : spécifie le type, arm7, arm9 etc..
  - <endianess> : organisation de la mémoire (big ou little)
  - <reset_mode> : spécifie ce que doit faire la cible après un reset

Exemple : target arm920t little reset_halt arm920t

Annexe 1.3 le fichier OpenOCD.cfg

gdbCmds

Ce fichier regroupe les commandes à envoyer au debugger. A noter que dans le test ci dessous les commandes sont envoyées à la main mais au final, ce fichier sera traité automatiquement. Voici les commandes contenues dans ce fichier :

- target remote localhost:3333 : connexion au port relatif au debug. Celui-ci est spécifié dans le fichier de configuration d'*OpenOCD*
- monitor reset : reset de la carte
- monitor sleep 500 : le cpu suspend ses activités durant 500 ms
- monitor poll : récupère l'état de la cible, running, halt etc...
- monitor soft_reset_halt : reset correct du *program counter*
- monitor arm7_9 sw_bkpts enable : autorise les breakpoints software pour les cibles de type arm7 ou arm9.
- load : télécharge le programme dans la ram de la cible.
- break main : ajout d'un breakpoint au début de la fonction *main.*

Nous voyons apparaître deux types de commandes. Les commandes *gdb* comme *break main* permettant le debug du programme en lui même et les commandes commençant par *monitor*. Ces dernières permettent de configurer *OpenOCD* et de lui demander d'interagir sur la cible.

Marche à suivre

Partons du principe que nous possédons un exécutable au format *.efl* correctement généré, avec les informations liées au debug. Comment allons-nous procéder pour le télécharger sur la carte et le debugger ? Pour ce faire nous allons tout d'abord alimenter la carte, brancher la *JTAG Key* à son interface *JTAG* puis démarrer *OpenOCD* en lui passant en paramètre le fichier de configuration adéquat. Nous allons ensuite démarrer *arm-elf-gdb* et entrer les commandes nécessaires les unes après les autres.

Exécution d'OpenOCD

Voici la fenêtre d'OpenOCD lorsque celui-ci est démarré correctement :

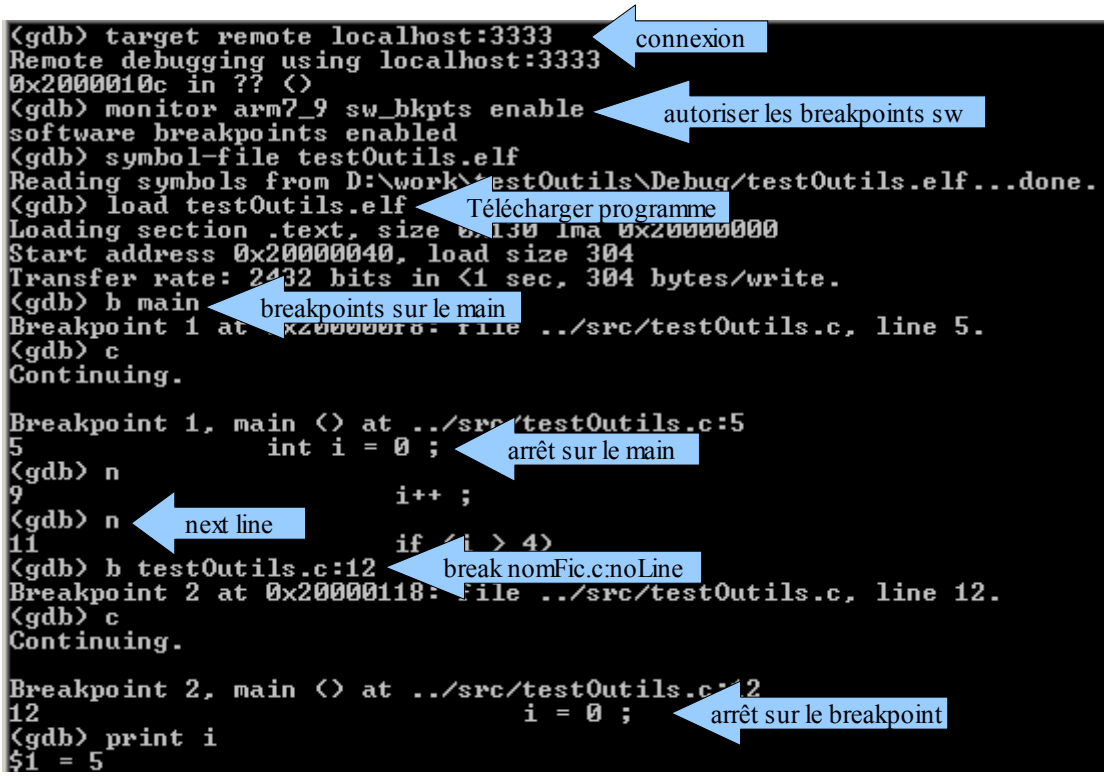
```
D:\>openocd-ftd2xx.exe -f openOCD.cfg
Info:    openocd.c:93 main(): Open On-Chip Debugger (2007-09-05 09:00 CEST)
```

Utilisation du debugger

Lorsque la connexion entre le PC et la carte est établie nous allons démarrer *arm-elf-gdb* pour envoyer les commandes nécessaires au debug de la carte. A noter que lorsque le plugin sera développé le fichier *gdbCmds* contiendra ces commandes. Elles seront ainsi exécutées automatiquement. Il faut bien entendu relever le port utilisé par *openOCD* pour transcrire les

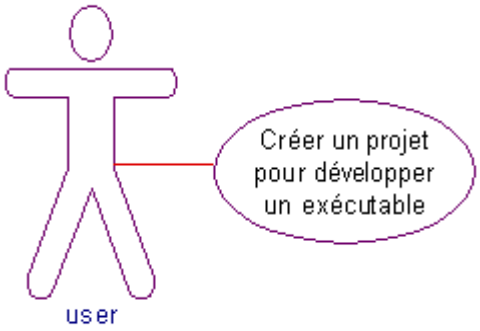
commandes *gdb* en commande *JTAG*. Celui-ci est spécifié dans le fichier de configuration d'*OpenOCD*.

Exemple

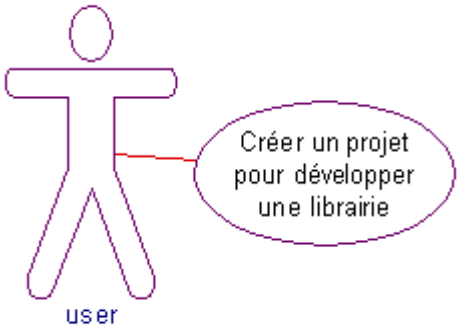


Cas d'utilisations

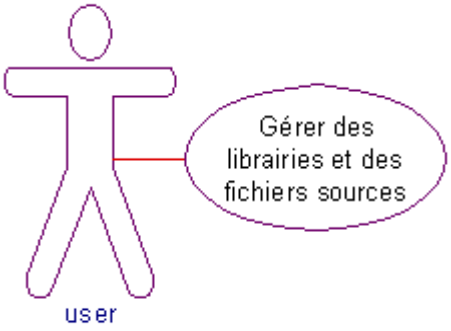
Pour présenter les fonctionnalités mises à disposition par le plugin *Embedded Hevs* une petite étude UML se résumant aux cas d'utilisation aborde les aspects principaux.



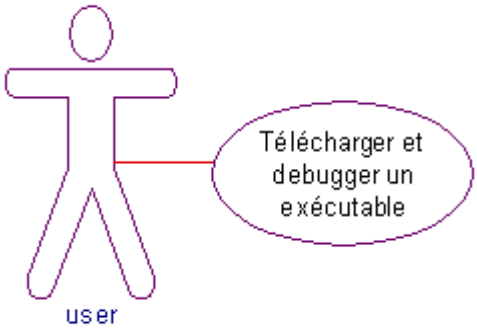
Le plugin doit permettre de créer un nouveau projet *standalone* spécifique à l'ARMEBS3. La création du projet doit être la plus simple possible.



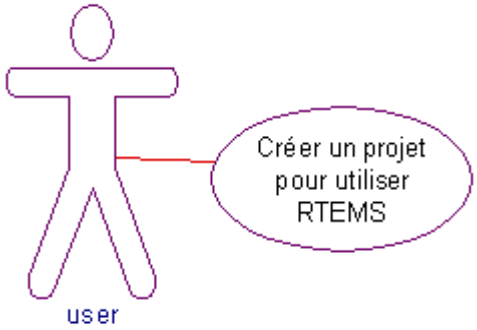
Le plugin doit permettre de créer un nouveau projet permettant de développer une librairie utilisable par la suite dans un projet de type exécutable



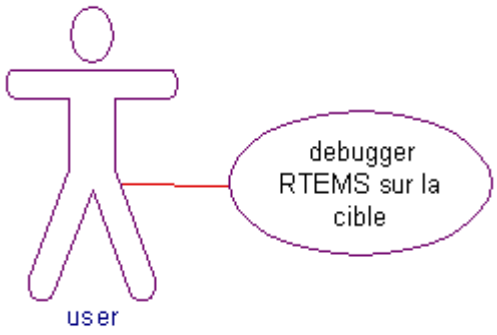
Comme pour un exécutable C standard il est possible de lui ajouter des fichiers sources et des libraires (spécifique à la cible) via la configuration du projet. A noter que nous reprenons la gestion des sources du plugin CDT.



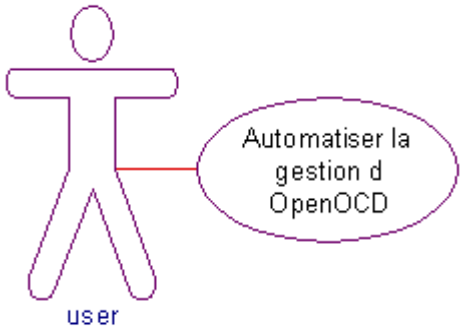
Quand un exécutable est correctement compilé et linké il est possible de le télécharger sur la carte et de l'exécuter en debug sans devoir modifier quoi que ce soit.



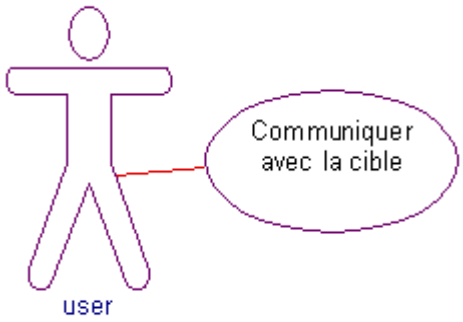
Il sera possible de créer un projet de type *RTEMS*. Ce dernier permettra d'utiliser les outils mis à disposition par *RTEMS* en chargeant un makefile par défaut. L'utilisateur n'aura plus qu'a déclarer les tâches, sémaphores, timers dont il aura besoin



Comme pour un exécutable *standalone* il sera possible de télécharger et de debugger *RTEMS* sur la cible. L'utilisateur aura aussi la possibilité de debugger et de voir l'exécution des directives de l'OS.



La gestion de l'interface de communication *OpenOCD* est entièrement gérée par le plugin. A noter qu'une console permet de récupérer les informations d'*OpenOCD* et de les afficher à l'utilisateur.

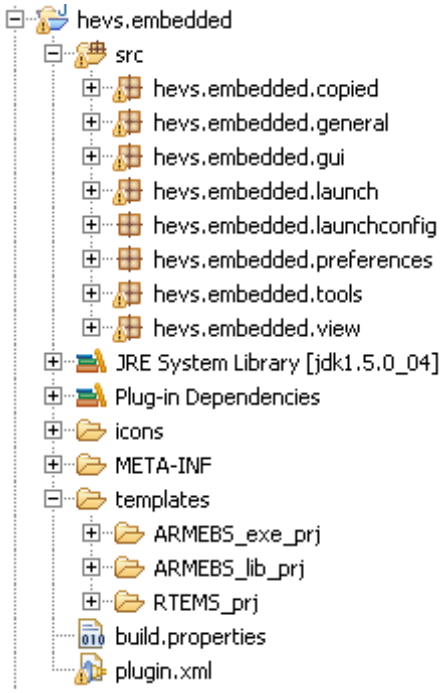


Pour faciliter la communication avec la cible un mini terminal intégré à éclipse permet de visionner l'output et d'envoyer des données sur la cible à l'aide du port série.

Développer un plugin

Avant de commencer le développement du plugin à proprement dit nous allons passer en revue l'organisation du workspace d'Eclipse ainsi que les outils mis à dispositions.

Organisation d'un projet



- **Sources (src)** : les sources du projet (le code). On peut constater que les plugins sont organisés à l'aide de plusieurs paquets.

- **Dépendances** : ce dont a besoin le plugin pour fonctionner (imports, points d'extensions...)

- **templates** : contient les templates utilisés lors de la création de nouveaux projets. Les sous répertoires contiennent les fichiers de base (*.ld, crt.s*...)

- **build.properties** : description du fichier JAR. Utilisé lors de l'exportation du plugin.

- **plugin.xml** : fichier principal du plugin. Il regroupe toutes les informations relatives au plugin, extentions, classes modifiées, fonctionnalités ajoutées...

Le fichier plugin.xml

```
<extension
         point="org.eclipse.cdt.core.templates">
      <template
            id="hevs.executable.armebs3.template"
            location="templates/ARMEBS_exe_prj/template.xml"
            projectType="org.eclipse.cdt.build.core.buildArtefactType.exe">
      </template>
      <template
            id="hevs.staticLib.armebs3.template"
            location="templates/ARMEBS_lib_prj/template.xml"
            projectType="org.eclipse.cdt.build.core.buildArtefactType.staticLib">
      </template>
```

Le fichier *plugin.xml* comporte plusieurs centaines de lignes ! Il serait très fastidieux de devoir l'éditer à la main pour modifier le plugin CDT ou pour ajouter une fonctionnalité au plugin. En guise d'exemple, l'extrait du ce fichier ci-dessus permet d'ajouter des templates au plugin.

Grâce au PDE, éclipse nous permet de simplifier l'écriture du fichier *plugin.xml*. Il est composé de plusieurs onglets regroupant toutes les informations nécessaires au développement du plugin :

- **Overview** cet onglet regroupe les informations générales relatives au plugin. Il est composé de plusieurs sections notamment celle de test permettant d'exécuter (et de debugger !) une deuxième instance d'éclipse avec le plugin développé à son bord. La section *Exporting* permet d'exporter le plugin dans un fichier .jar pour pouvoir l'installer sur d'autres machines.
- **Dependencies**

Cet onglet permet de spécifier quels packages vont être utiles pour le développement du plugin.
Plusieurs dépendances sont associées à notre plugin notamment les paquets relatifs à CDT. En effet nous avons pu constater plus haut dans le fichier plugin.xml que notre plugin allait ajouter des templates, il faut donc ajouter cette dépendance pour pouvoir modifier les classes s'occupant de cette configuration.
De plus, notre plugin va gérer le debug à travers le port *gdb* il faudra donc aussi modifier les classes accédant à la configuration du debuggeur.

- **Extensions**

  L'onglet *Extensions* est la partie centrale du plugin. C'est ici qu'on va définir l'architecture du plugin (les vues, les menus, les actions...). C'est surtout l'endroit où l'on va déclarer dans quelle partie d'Eclipse on va brancher notre plugin. Pour notre plugin plusieurs extensions sont nécessaires. L'extension *ManagedBuildInfo* est la plus importante pour nous elle permet de modifier la configuration d'un projet *C* lors de sa création. Elle regroupe plusieurs éléments comme la configuration du linker de l'assembleur et possède une sous-extension qui gère la configuration de l'exécutable pour l'ARMEBS3. Nous allons voir plus en détail les extensions ajoutées pour notre plugin

La suite du rapport décrit en détail les extensions déclarées pour développer le plugin *Embedded Hevs.*

Les extensions « New Project »

Pour commencer

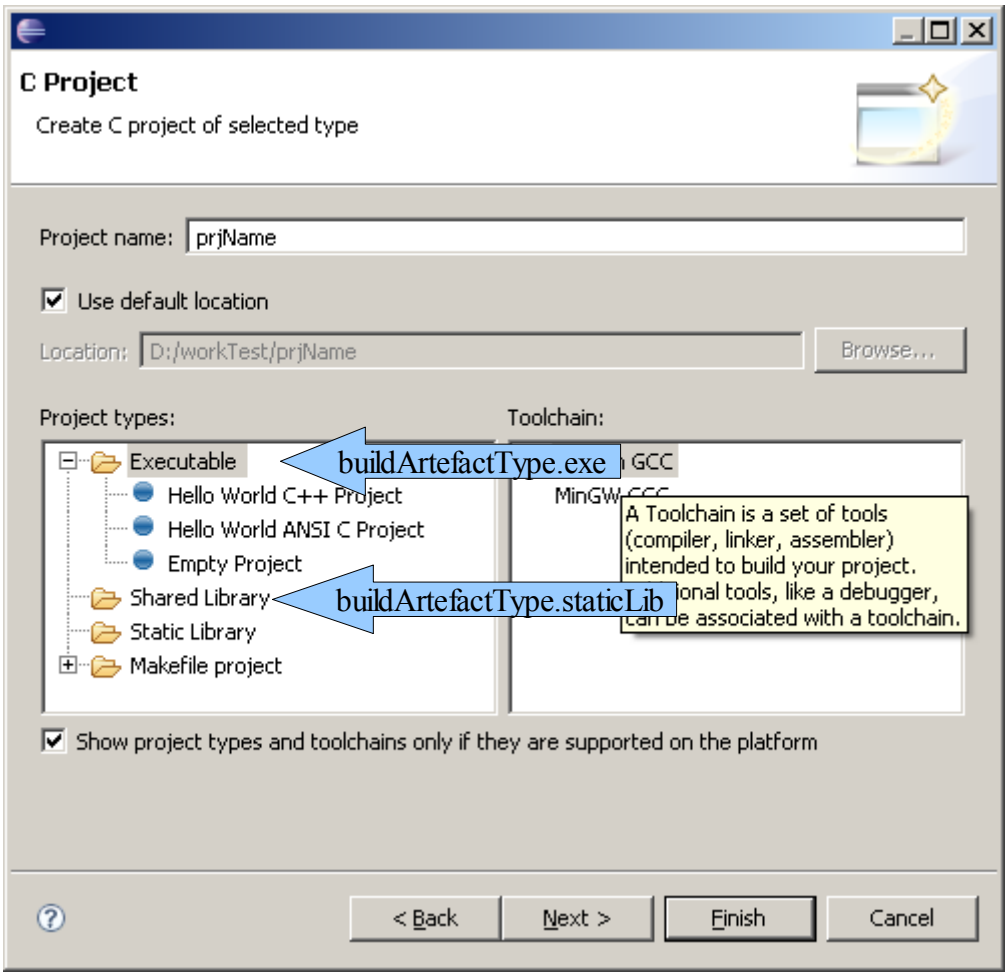
Dans cette section, nous allons aborder deux aspects assez importants introduit depuis CDT 4.0. Ces derniers apparaissent tout au long de la création des extensions nécessaires à la déclaration de *toolchains, configuration* et *projectType.*

Les types de projets (buildArtefactType)

Le nouveau mécanisme de type de projet introduit par CDT 4 regroupe trois types de projets de base :

- org.eclipse.cdt.build.core.buildArtefactType.exe : un exécutable
- org.eclipse.cdt.build.core.buildArtefactType.staticLib : une libraire statique
- org.eclipse.cdt.build.core.buildArtefactType.sharedLib : une librairie dynamique

On entend par *buildArtefactType* la nature de l'objet qui sera construit par le type de projet *C*. Le wizard permettant de créer un nouveau projet *C* regroupe le type de projet dans le dossier associé au *buildArtefactType* et lui associera une *toolchain* si celle-ci est compatible.

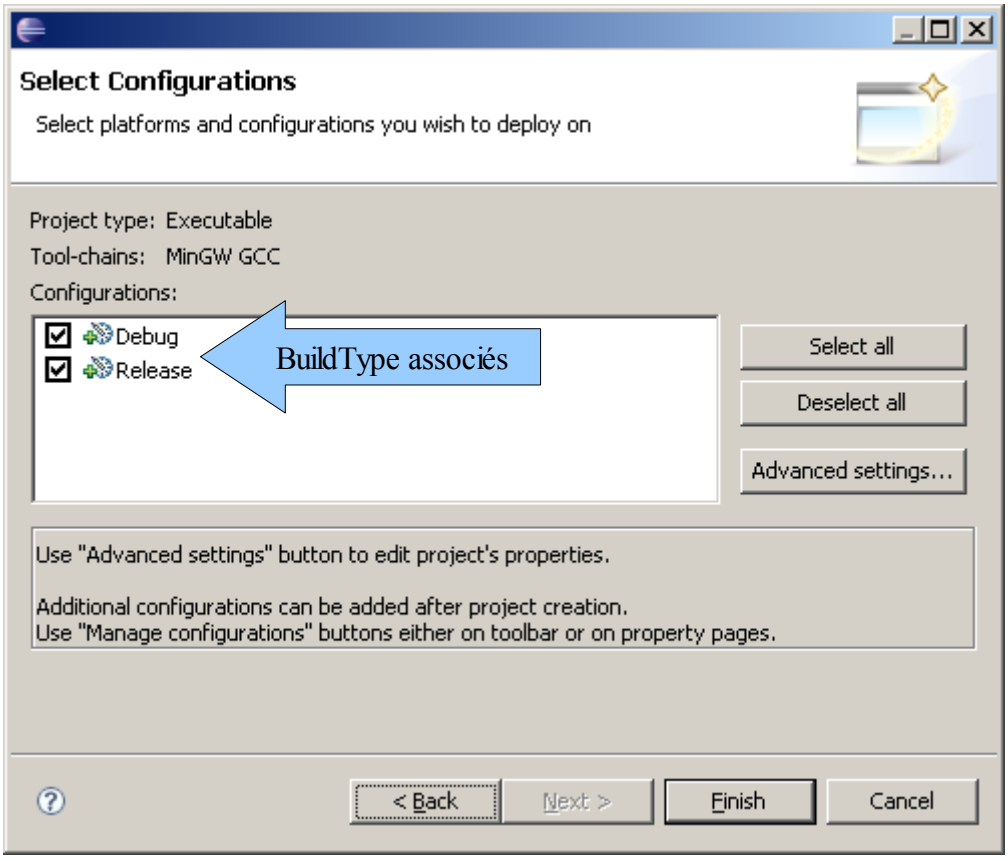


Les type de configuration (buildType)

Par type de configuration on entend la façon dont le projet va être compilé, dans la plupart des cas deux types sont utilisés :

- org.eclipse.cdt.build.core.buildType.debug : exécutable avec informations de debug
- org.eclipse.cdt.build.core.buildType.release : exécutable sans informations de debug

Cette façon de faire est très pratique, on peut modifier les flags relatifs au compilateur de manière à générer un exécutable supportant le debug ou non. A noter que d'autres flags peuvent varier suivant la configuration du projet, l'optimisation par exemple.



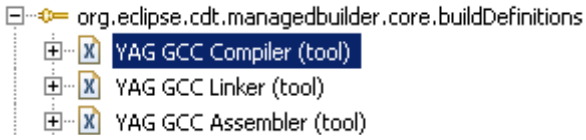
org.eclipse.cdt.managedbuilder.core.buildDefinitons

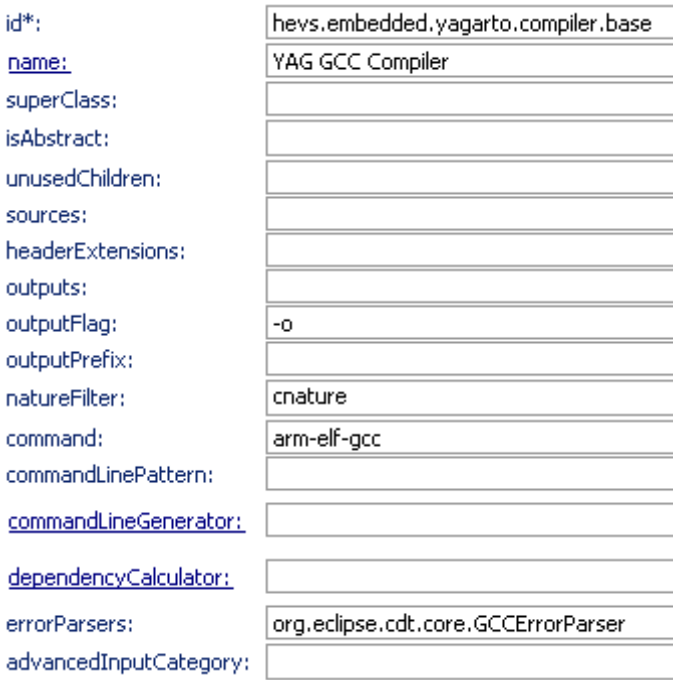
Cette extension va nous permettre d'ajouter des *toolchain*s qui seront mises à disposition lors de la création d'un nouveau projet de type *C*. Pour ce faire nous allons déclarer un certain nombre de sous extensions relatives à un projet. Pour mieux comprendre toute l'organisation de cette extension, nous allons énumérer les différents outils proposés.

Tool

On entend par *tool* un outils lié à une commande (*gcc, as*...) spécifique. Un *tool* représente par exemple un compilateur ou un linker celui-ci peut avoir un grand nombre d'options associées, notamment au niveau du look lors du déploiement du plugin. En principe chaque *tool* a un *input* et un *output* par exemple un compilateur prend comme entrée des fichier *.c* et génère des objet *.o* en guise de sortie.

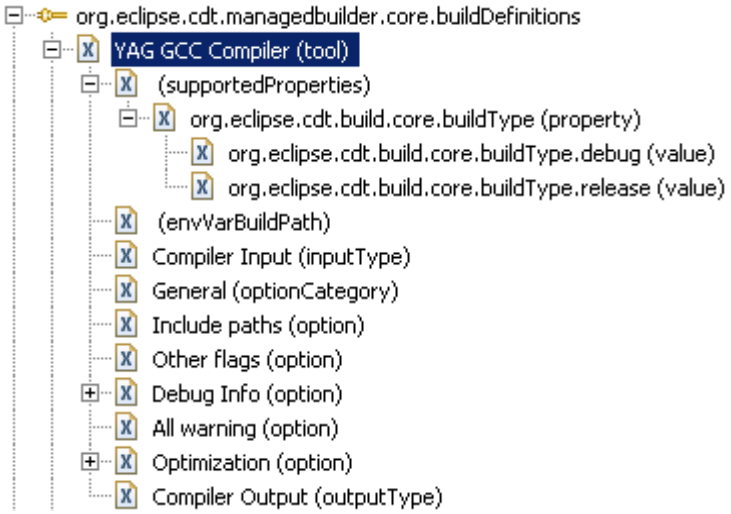
Exemple 1, le compilateur



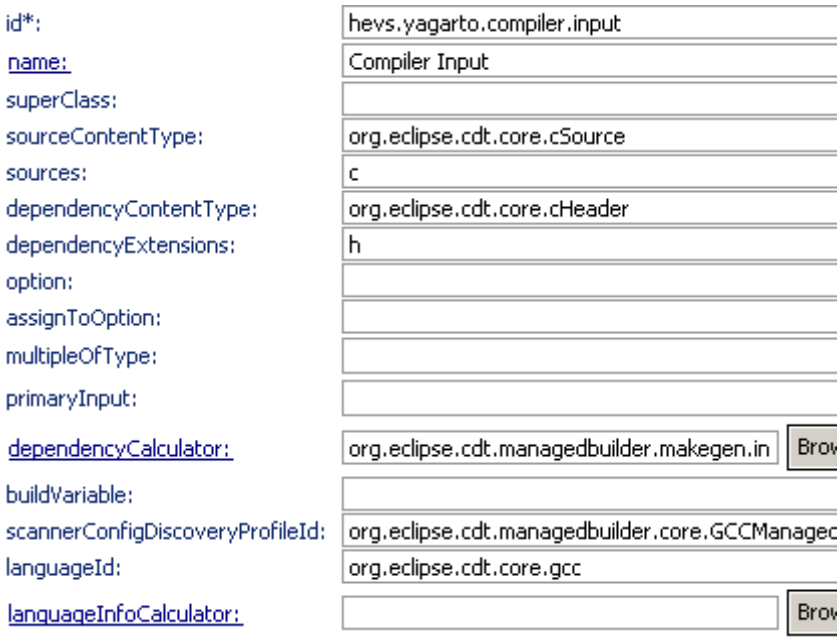


Ci-contre on peut voir les options générales du compilateur. Celui-ci est défini par un identifiant unique et par un nom qui apparaîtra dans les settings du projet *C*. Il faut en outre lui définir un *outputFlag* et une commande associée. L'option *natureFilter* spécifie que ce *tool* est définit pour les fichiers de type *C* (et pas C++).

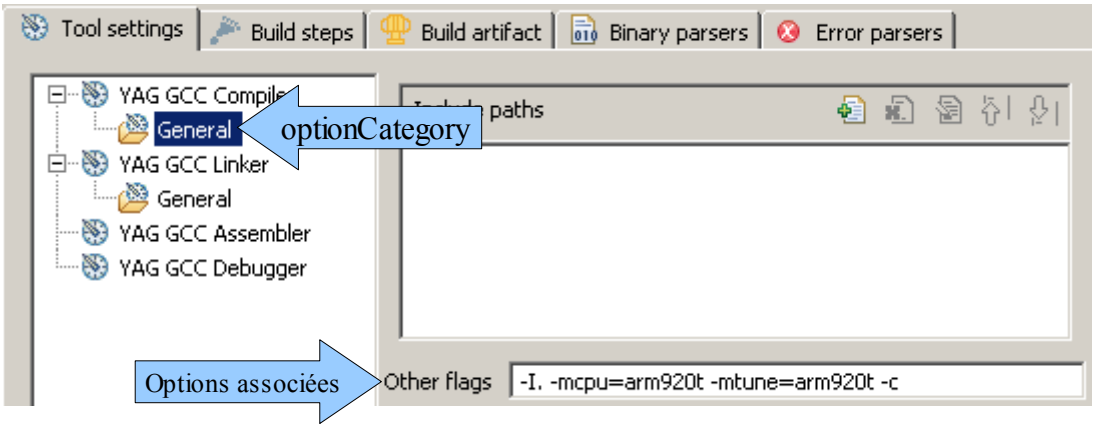
Plusieurs options apparaissent lorsque l'on développe le *tool*.



- ***SupportedProperties*** par cette option on spécifie que le compilateur supporte les deux types de configurations, à savoir *debug* et *release*. Cette option nous permettra de différencier les flags utilisés dans les deux cas. Le linker lui ne différencie pas ces deux types.
- ***InputType*** cette option permet de définir précisément quels types de fichiers vont être utilisés par le compilateur en guise d'entrée. On peut voir que les sources sont représentées par leur extension *.c* et *.h* et sont définies avec les identifiants *org.eclipse.cdt.core.cSource et .cHeader*.



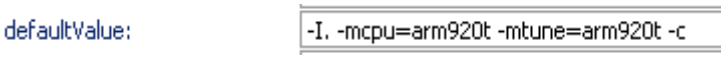
- ***OptionCategory*** cette catégorie permet de regrouper un certain nombre d'options. Elle est principalement utilisée pour le look de la configuration du Projet *C*. Elle est définie par un identifiant unique, un *owner (*l'id du compilateur), on peut aussi lui associer un nom et une icône.



- ***L'option Other flags*** ajoute des flags qui seront utilisés par défaut par le compilateur voici une brève description de leur utilité.

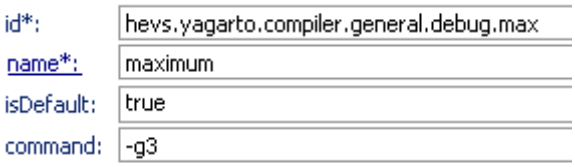
1) -mcpu=arm920t : configure *GCC* pour le processeur spécifié
2) -mtune=arm920t : optimisation pour le processeur spécifié
3) -c : seulement générer les fichiers objets. Ne pas linker.

Cette valeur est spécifiée dans l'option ci-dessous :

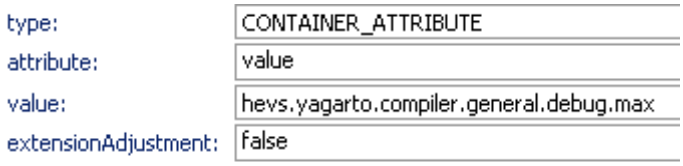


- ***L'option Include paths*** représente une widget permettant d'ajouter des dossiers que le compilateur devra scruter si les fichiers sources utilisés ne se trouvent pas dans le dossier *root*. La rubrique *valueType* permet de spécifier le type d'option (une simple *textBox*, une liste d'informations, une *combobox* etc...). La rubrique *browseType* permet de spécifier le type de *browser* associé à l'option : un *browser* de fichier, de répertoire... On spécifie encore la commande utilisée (ici le -I) qui sera utilisé dans le *makefile* pour la compilation des fichiers.

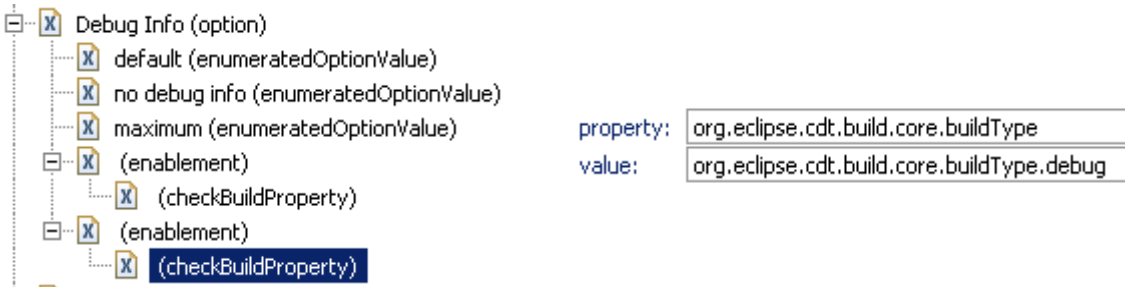
- ***L'option Debug Info*** est intéressante car elle met en relation les *buildType* définis plus haut. Dans le projet *C,* cette option est représentée par une *combobox* regroupant les flags relatifs au debug (*-g, -g1, -g2*) permettant d'ajouter ou non les informations de debug. Cette option est directement associée au *buildType.* En effet si on compile notre exécutable avec la configuration *release* il est inutile d'inclure les informations de debug. On remplit la *combobox* en ajoutant un certain nombre d'élément (*enumeratedOptionValue)*. A chaque élément est associé un identifiant, un nom et une commande. Cette commande sera utilisée dans le *makefile* lors de la construction des fichiers objets.



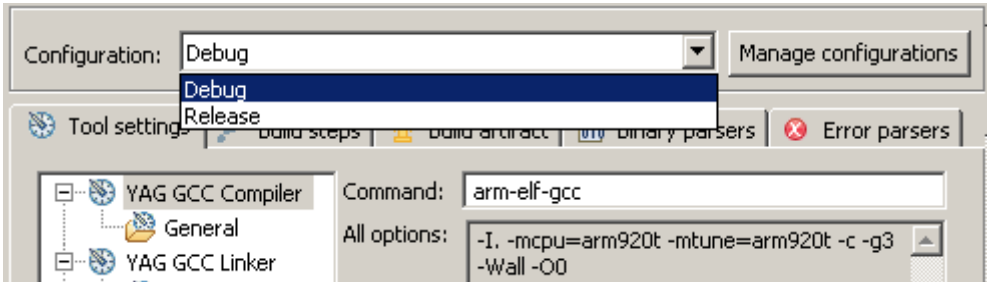
Pour choisir quelle commande associer à quelle type de configuration il faut ajouter une sous option nommée *enablement* contenant entre autre l'identifiant du flag :



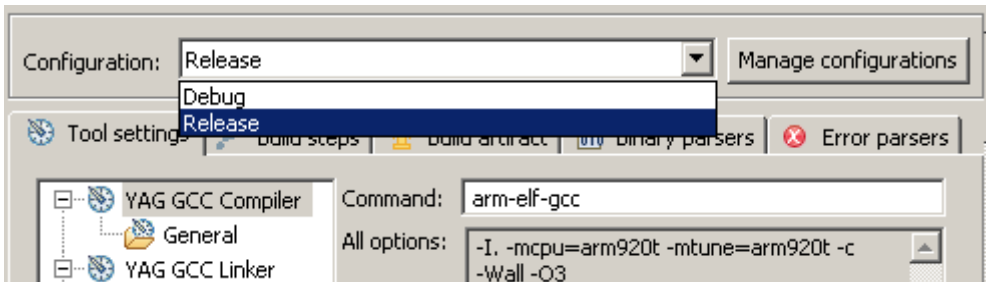
pour que cette option soit utilisée par défaut par une configuration de type *debug* il faut rajouter à cette sous option une rubrique *checkBuildProperty* permettant de spécifier le *buildType* associé. Dans cette exemple le flag *-g3* est associé à une configuration de type *debug* il faut donc spécifier *org.eclipse.cdt.build.core.buildType.debug.*



Lorsque l'on teste le plugin, on peut constater que les flags par défaut changent dès lors qu'on sélectionne *debug* ou *release* en guise de configuration.

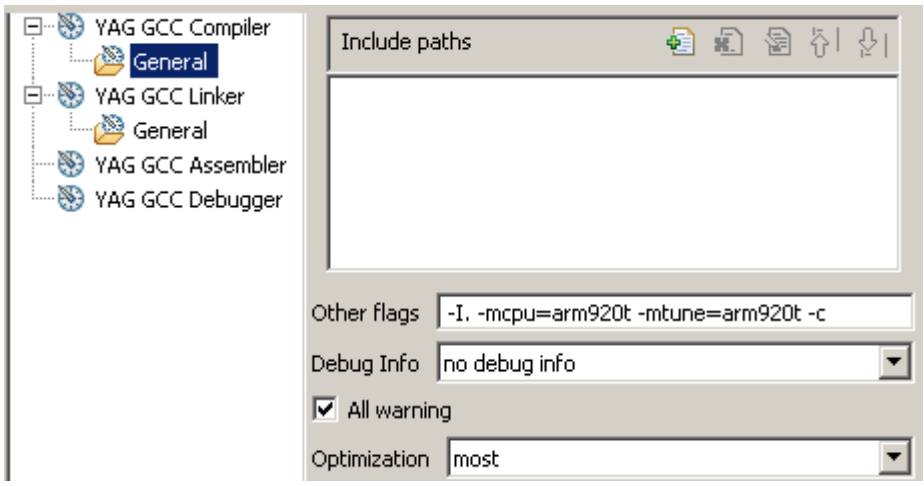


La configuration *debug* sélectionne les flags *-g3* et *-O0* (optimisation)

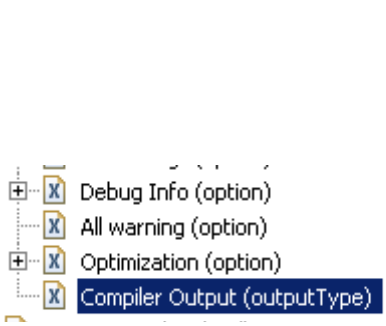


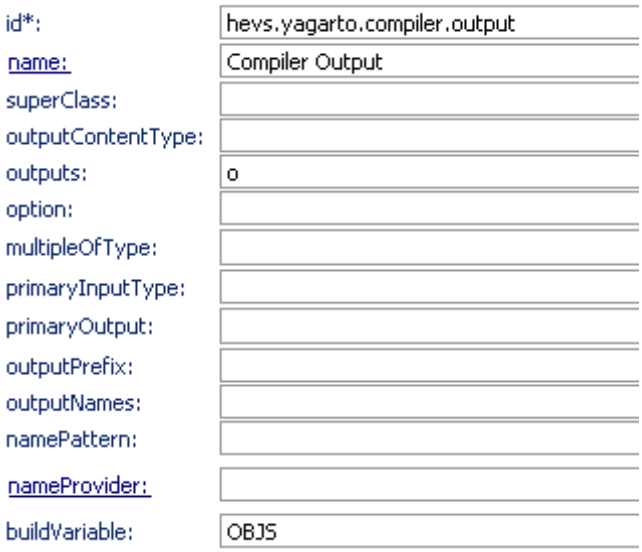
La configuration *release* enlève le flag concernant le debug et modifie le flag relatif à l' l'optimisation *-O3*.

Il est aussi possible de modifier ces options dans l'*optionCategory* « General »



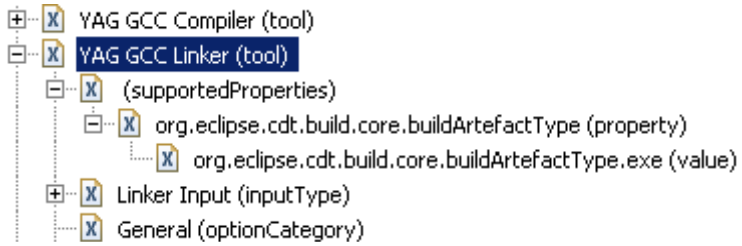
- ***outputType*** représente la sortie d'un *tool*. Par sortie on entend le type de fichier qui sera généré par la commande associée au *tool*. Dans notre exemple la commande associée au compilateur va compiler des objets *.o* et ces objets seront listés dans une *buildVariable* nommée *OBJS*. Cette variable sera utilisée dans le *makefile* généré automatiquement.





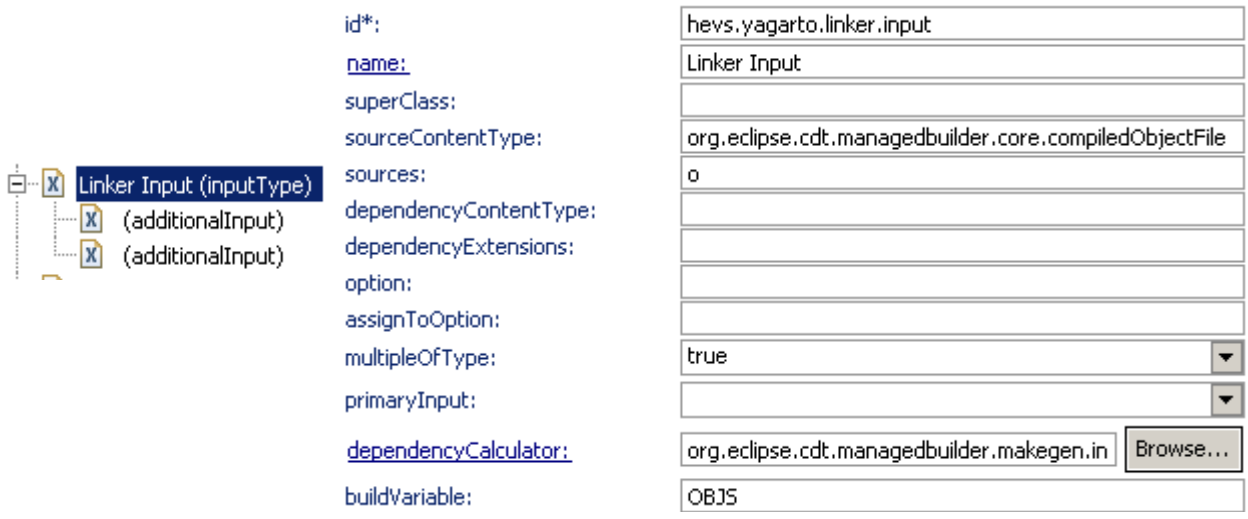
Exemple 2, le linker

le linker est aussi considéré comme un tool. Tout comme le *compilateur* il a un *nom*, une *commande,* un *outputFlag* etc... Cependant, la deuxième nouveauté introduite plus haut fait son apparition avec le linker. En effet, le plugin *Embedded Hevs* permet de compiler un exécutable standard mais aussi de compiler des objets et de les archiver dans un librairie *(.a)*. On constate donc que le linker n'a aucun travail à faire pour un projet de type *buildArtefactType.staticLib* et cette option peut être spécifiée dans les extensions en ajoutant une *supportedProperties* de type *buildArtefactType.exe :*



L'archiveur possède lui aussi une *supportedProperties* mais la valeur de celle-ci sera *buildArtefactType.staticLib.* En effet, l'archiveur est, en principe, utilisé pour construire une librarie et n'a rien à faire dans la construction d'un exécutable.

Pour bien comprendre la notion de *toolchain* associée à ces extensions voyons encore le type d'entrée que reçoit le linker. Comme pour le compilateur l'*inputType* reçoit un identifiant unique, il faut en outre spécifier le type de sources associées (les objets *.o*) et notion plus important, le champ *buildVariable* contient la variables associée aux objets spécifiés par l'*outputType* du compilateur. Le *makefile* sait alors où aller chercher les objets lorsque l'application sera linkée. L'*inputType* du linker possède encore deux entrées additionnelles à savoir la variable *$(USER_OBJS)* représentant les objets supplémentaires pas forcément liés au projet et la variable *$(LIBS)* contenant les noms des librairies ajoutées par l'utilisateur.

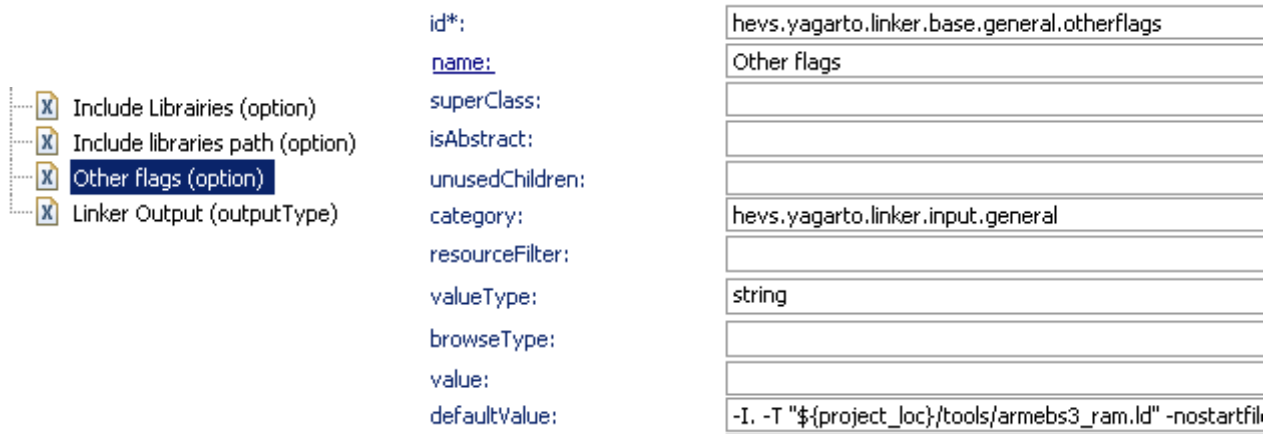


Comme pour le compilateur, le linker possède aussi une option *otherFlag* celle-ci ajoute des flags qui seront utilisés par défaut par le linker voici une brève description de leur utilité.

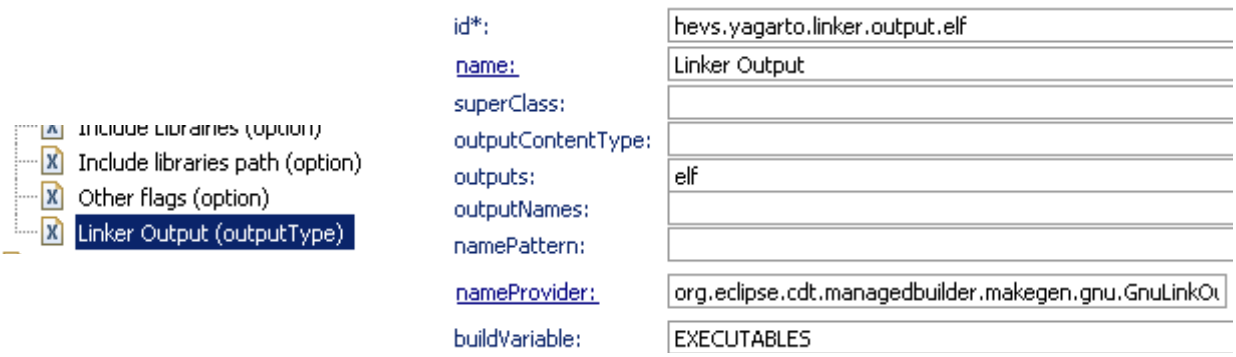
1) -I. : permet d'inclure le dossier courant
2) -T *monFichierDeLien.ld* : spécifie le fichier de liens (explications dans le chapitre *Debugger un .elf avec ces outils)*
3) -nostartfiles : indique de ne pas utiliser le fichier c*rt0.s* par défaut et de le remplacer par celui spécifié. (explications dans le chapitre *Debugger un .elf avec ces outils)*
4) -Map=*fichierDeMap.map* : construit un fichier représentant le mapping de la mémoire.

A noter que le fichier *.ld* est différent pour chaque projet *C*. C'est pourquoi la variable interne d'éclipse *${project_loc}* représentant le path du projet est utilisée pour spécifier le nom complet du fichier *.ld* utilisé :

${project_loc}/tool/armebs3_ram.ld = *D:/work/prjTest/tool/armebs3_ram.ld*



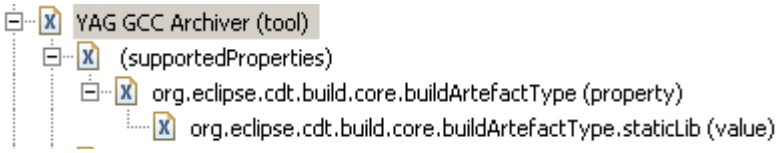
Parlons maintenant de l'*outputType* du linker. Bien entendu le linker va lier les objets ensemble et générer ensuite l'exécutable portant l'extension *.elf*. Ces informations doivent être spécifiées dans les extensions :



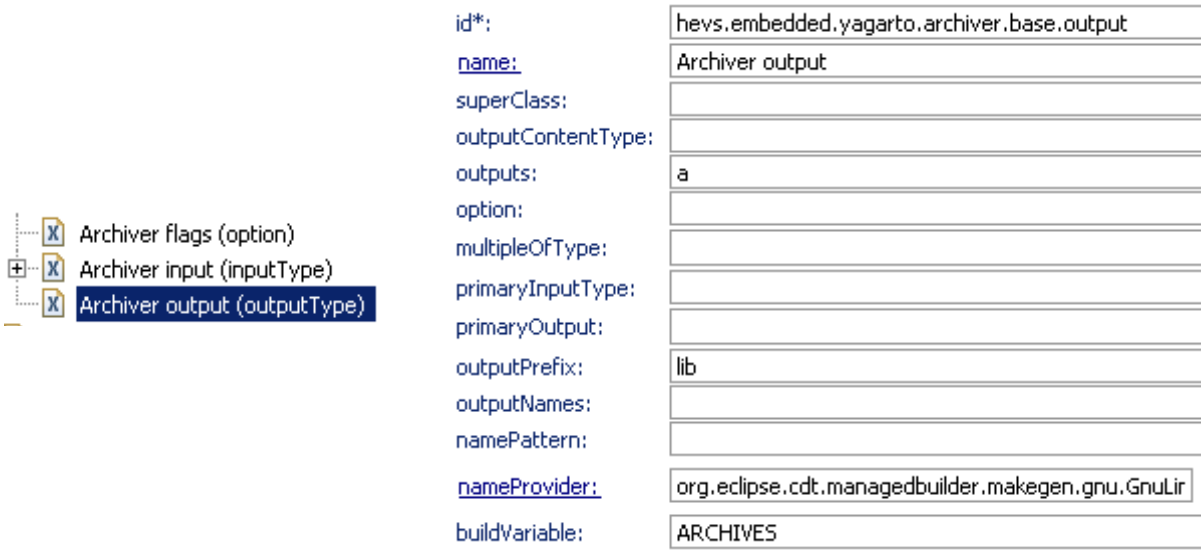
Notez que le champs *buildVariable* informe le *makefile* que l'output de ce *tool* est un exécutable.

Exemple 3, l'archiveur

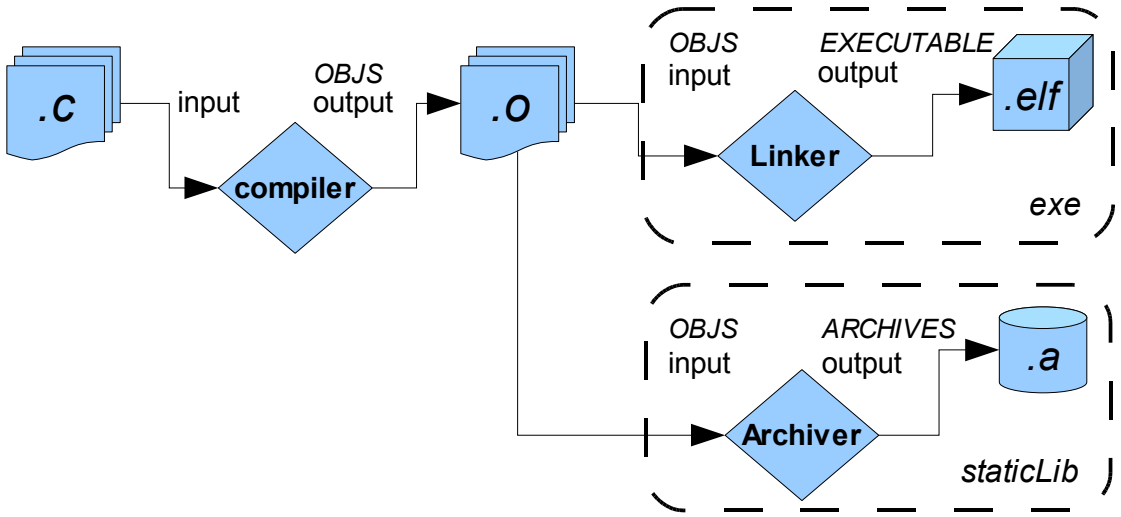
Voyons maintenant rapidement comment l'archiveur est déclaré dans les extensions. Cela va nous permettre de mieux comprendre comment les types de projets (*staticLib* et *exe*) sont gérés. Tout d'abord, ce *tool* est déclaré comme n'importe quelle autre avec un *id,* une *commande (arm-elf-ar)* et les informations de base. On constate ici que l'archiveur possède une *supportedProperties* avec la valeur *buildArtefactType.staticLib.* En effet, l'archiveur est, en principe, utilisé pour construire une librairie.



L'archiveur possède exactement le même *inputType* que le compilateur, en effet une archive regroupe les objets compilés ainsi que les librairies utilisées. C'est au niveau de l'*outputType* que les principales différences apparaissent : au niveau de l'extension (*.a)*, du préfixe *(lib)* et du champs *buildVariable* qui spécifier au *makefile* que la sortie est une archive et non un exécutable.

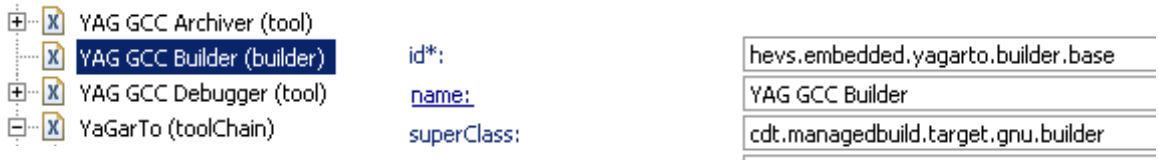


Pour finir la présentation des *tools,* voici une petite illustration résumant les notions abordées.



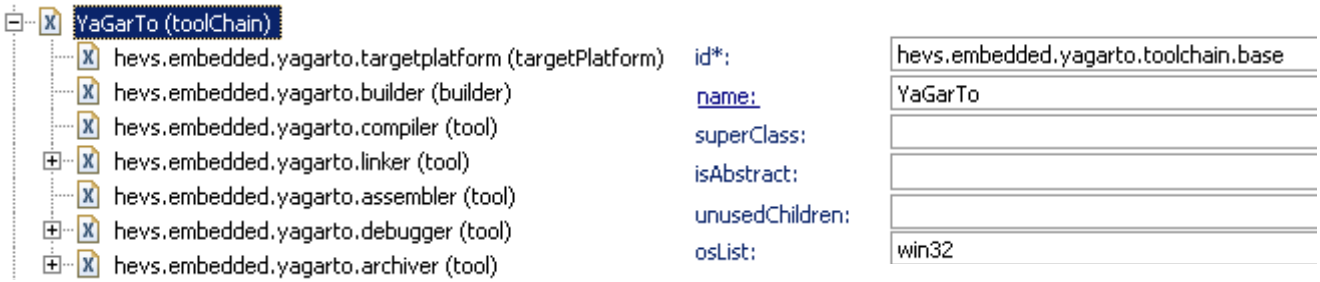
Builder

Le *builder* est simplement l'outil qui, associé au *makefile,* va généré l'exécutable ou l'archive relatif au projet *C*. Dans notre cas nous avons simplement à hériter du *builder* de GNU *(make)* déclaré par le plugin *CDT*. Pour l'héritage il suffit de recopier l'identifiant de la classe parente dans le champs *superClass*.



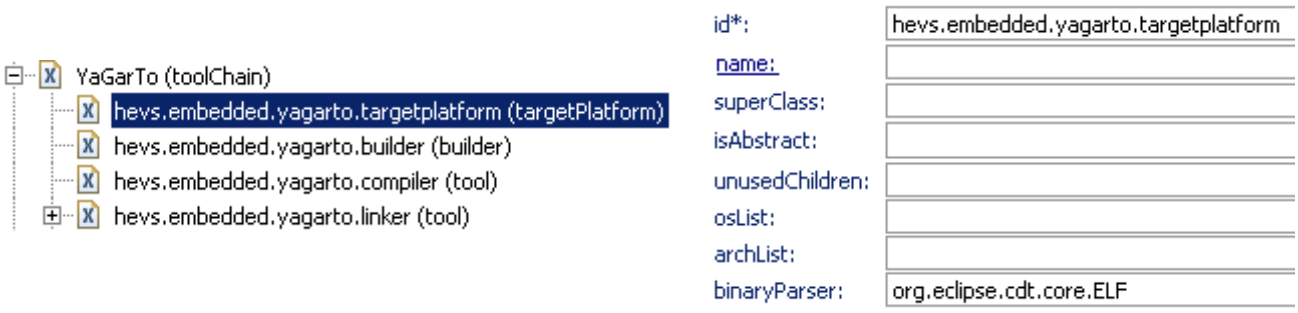
Toolchain

On entend par *toolchain* un ensemble d'outils permettant de compiler, lier et debugger un exécutable. Dans les extensions du plugin, une toolchain regroupe simplement les *tools* déclarés plus haut. Il suffit ensuite, comme pour le *builder*, de déclarer les *tools* associés à la *toolchain* et de spécifier les classes parentes dans le champs *superClass.*



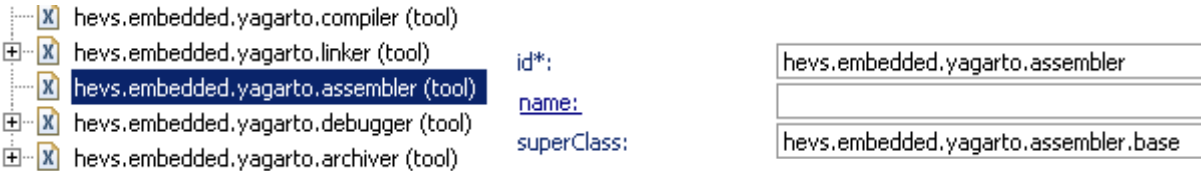
Pour déclarer la *toolchain* il suffit de lui donner un id, un nom et de préciser l'OS sur lequel le plugin va s'exécuter.

On peut aussi constater qu'une nouvelle sous extension apparaît ici : la *targetPlatform,* elle détermine la plat forme qui accueillera l'exécutable ou l'archive générée.



La notion la plus importante est le *binaryParser.* On spécifie ici que le format de sortie du projet *C* et le format *elf.*

Parlons maintenant des *tools* de notre *toolchain,* on peut voir qu'on retrouve exactement les outils déclarés plus haut à savoir un *builder*, un *compilateur*, un *linker*, un *assembleur*, un *debugger* et un *archiver*. Pour que les *tools* déclarés plus haut fassent partie de la *toolchain* il faut les ajouter et leur attribuer la *superClass* associée. Pour l'assembleur cela donne :



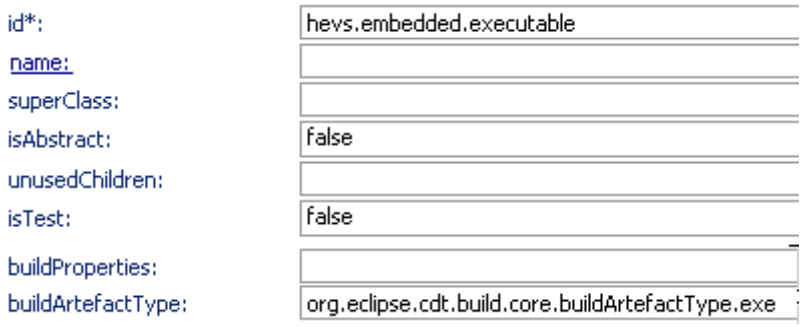
Cette manière de faire est identique pour tous les outils, cependant on peut constater que le *linker*, le *debugger* et *l'archiveur* possèdent une sous-extension. Cela paraît logique car *l'archiveur* n'est utilisé que pour des projets de type *buildArtefactType.staticLib* tandis que le *linker* et le *debugger* sont utilisé dans des projet *C* permettant de créer des exécutables à savoir des projets de type *buildArtefactType.exe.* Pour que le linker soit activé lorsque l'on développe un exécutable, il faut rajouter l'extension *enablement* et lui donner la valeur *buildArtefactType.exe* en guise de *buildArtefactType.* Idem pour le debugger.

Pour l'archiveur la manière de faire est similaire, cependant le type n'est plus *buildArtefactType.exe* mais *buildArtefactType.staticLib.*

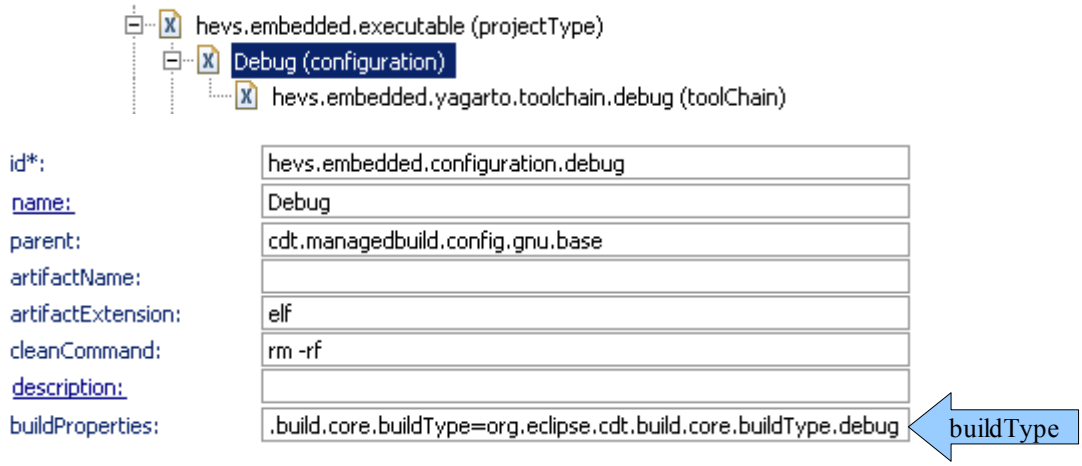
ProjectType

Cette extension représente un type de projet en lui associant un *buildArtefactType* (*exe* ou *staticLib*) des configurations *(debug* et *release)* ainsi qu'une *toolchain.*

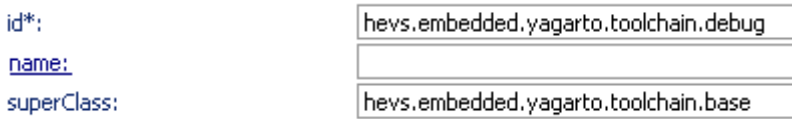
Pour définir un type de projet il faut lui associer un id et d'autres informations, mais surtout lui attribuer le bon *buildArtefactType*, à savoir *buildArtefactType.exe* pour un exécutable et *buildArtefactType.staticLib* pour une librairie.



Il faut maintenant ajouter au type de projet les deux configurations que notre plugin peut supporter, à savoir *debug* et *release*. De cette façon les flags associés aux deux configurations seront automatiquement chargés. Il faut encore spécifier l'extension de l'exécutable ainsi que la commande utilisée pour « nettoyer » le projet. On précise encore le type de configuration en ajoutant l'information *buildType.debug* à la variable *buildType* dans le champs *buildProperties*.

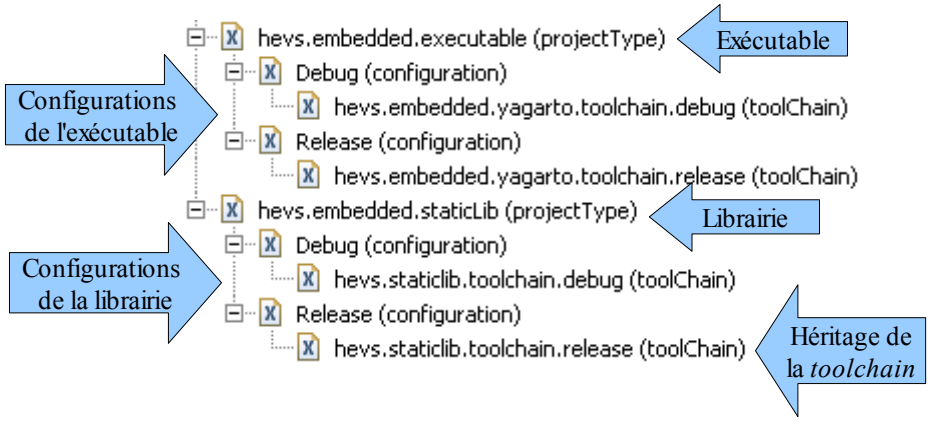


Pour finir il faut associer une *toolchain* à la configuration. Pour ce faire nous allons simplement hériter de la *toolchain* déclarée au dessus.



La manière de faire pour la configuration *release* est identique à celle de *debug* il suffit de remplacer le *buildType.debug* par *buildType.release* dans le champs *buildProperties*

Aperçu des types de projets



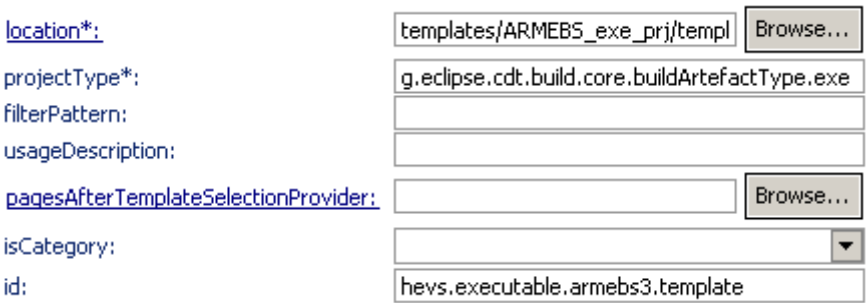
org.eclipse.cdt.core.templates

CDT 4.0 possède un framework permettant de développer des templates. Ces templates sont utilisés lorsqu'un nouveau projet *C* est créé. Ils permettent d'associer des fichiers et des dossiers de base qui seront automatiquement copiés lors de la création du projet. La configuration de ces templates est rédigée à l'aide d'un fichier *template.xml*

Pour créer un template il faut tout d'abord ajouter une extension en spécifiant les informations suivantes :

- le path du fichier *template.xml.*
- le type de project (*exe* ou *staticLib*) ainsi le template apparaîtra dans la bonne rubrique lors de la création d'un nouveau projet *C*.
- l'identifiant unique du template.

Aperçu de l'extension



Le fichier template.xml

Ce fichier regroupe les actions à réaliser lors de la création d'un nouveau projet. Un exemple assez complet est disponible dans l'aide officielle d'éclipse. En guise d'exemple nous allons parcourir quelques lignes du fichier.

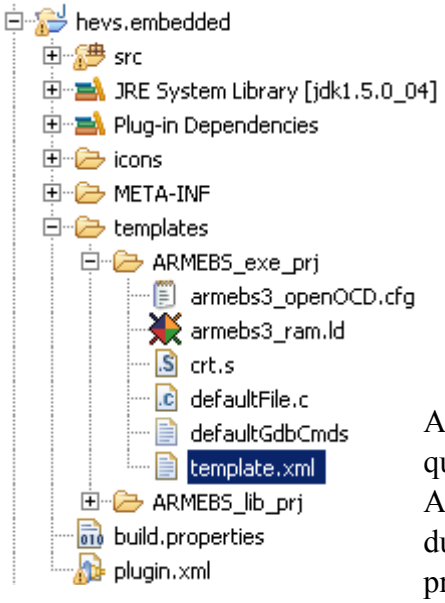
Il y a tout d'abord l'en-tête du fichier *XML* regroupant les informations de base du template, un identifiant, un label qui sera visible par l'utilisateur ainsi qu'une description.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<template   type="ProjTempl" version="1.0" supplier="Eclipse.org"
            revision="1.0" author="HEVS" copyright=""
            id="ARMEBSDefaultProject" label="Embedded ARMEBS3"
            description="Project for ARMEBS3 development board" help="">
```

Pour créer automatiquement un dossier qui contiendra les sources du projet *C* il faut utiliser la syntaxe *XML* suivante :

```
<process type="org.eclipse.cdt.core.CreateSourceFolder">
    <simple name="projectName" value="${projectName}"/>
    <simple name="path" value="${sourceDir}"/>
</process>
```

Pour ajouter des fichiers automatiquement en copiant les fichiers de base du plugin la syntaxe suivante est utilisée :




```
<process type="org.eclipse.cdt.core.AddFiles">
    <simple name="projectName" value="${projectName}"/>
    <complex-array name="files">
        <element>
            <simple name="source" value="defaultGdbCmds"/>
            <simple name="target" value="tools/gdbCmds"/>
            <simple name="replaceable" value="true"/>
        </element>
    </complex-array>
</process>
```

A gauche nous avons l'arborescence du plugin. On peut constater que les fichiers de bases sont tous regroupés dans un même dossier. A droite, la syntaxe permettra de copier le fichier *defaultGdbCmds* du plugin dans le dossier tools et portera le nom *gdbCmds* dans le projet *C*

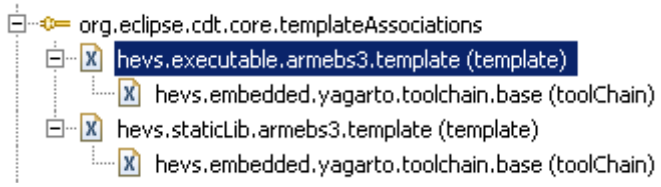
Le plugin *Embedded Hevs* déclare deux templates, un pour les exécutables et l'autre pour les librairies statiques. On peut cependant imaginer avoir un template pour chaque cible utilisée, il est possible ainsi d'avoir des fichiers de configuration et de commandes spécifiques pour chaque projet. Je passe assez rapidement sur cette notion de template car il est très facile de recopier le fichier *template.xml* pour ajouter un template ainsi que les fichiers de base pour la gestion d'une nouvelle cible. A noter qu'un exemple complet est disponible sur le site officiel d'éclipse.

Les fichiers de base sont décrits en détail dans le chapitre *Debugger un .elf avec ces outils*

Annexe 1.4 template.xml

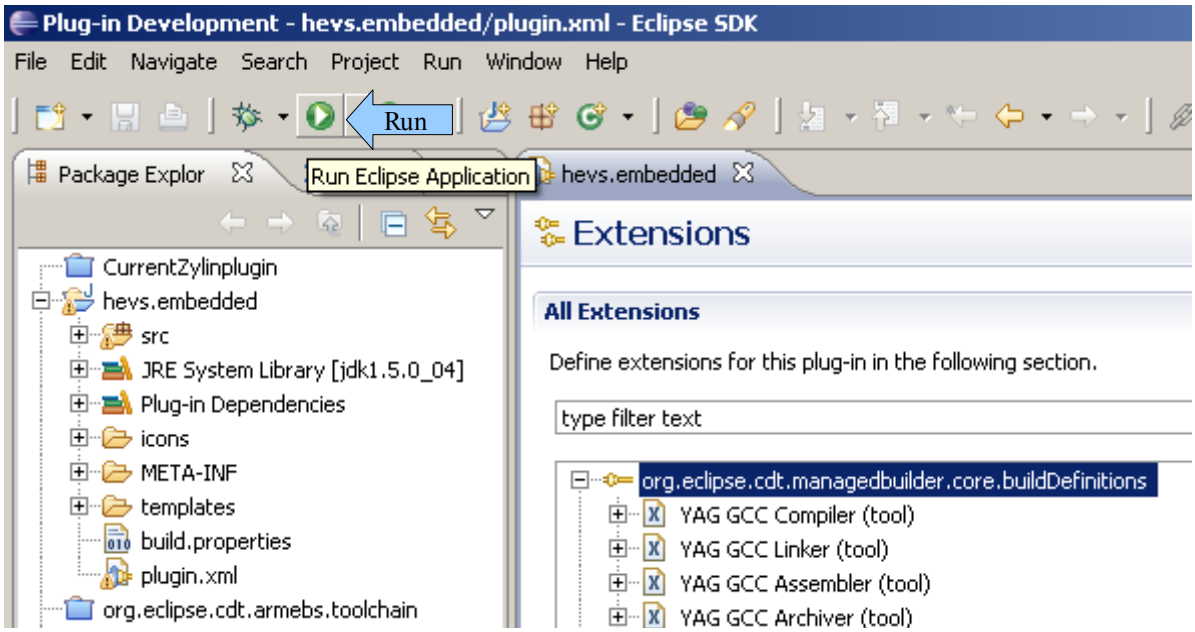
org.eclipse.cdt.core.templateAssociations

Cette extension permet de faire correspondre un *template* avec une *toolchain*. Il est bien entendu possible de lier plusieurs *toolchains* avec un *template*. Lorsque l'utilisateur choisit un type de projet dans le wizard *new Project C*, les *toolchains* à dispositions apparaissent à droite.

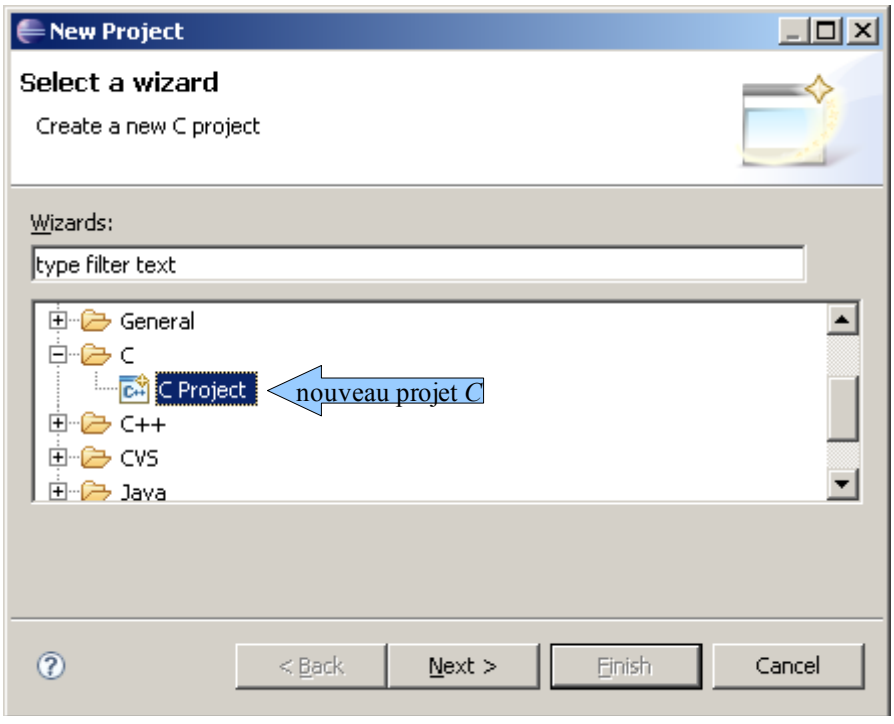


Test des extensions « New Project »

Après avoir développé les extensions nécessaires à la création et au management d'un projet *C*. Il va falloir tester les nouvelles fonctionnalités mises à disposition par le plugin *Embedded Hevs*. Tout d'abord il faut lancer une nouvelles instance d'éclipse avec le plugin développé à son bord.



Lorsque la nouvelle instance d'éclipse est démarrée il faut créer un nouveau projet de type *C*. Pour ce faire il faut cliquer sur *File – New – Project*. Puis dans la rubrique *C* il faut sélectionner simplement *C Project.*

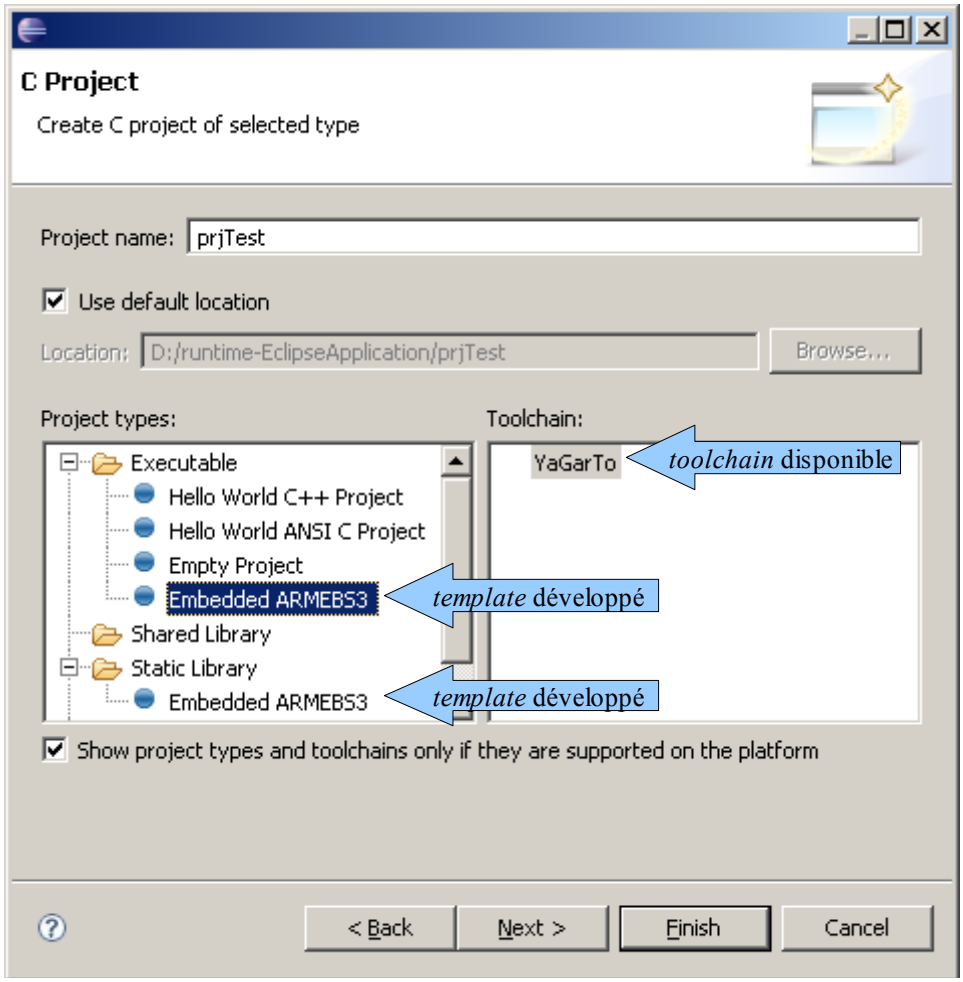


Pour continuer il faut cliquer sur le bouton *Next*. Apparaît ensuite la fenêtre la plus importante pour la création d'un nouveau *projet C*.

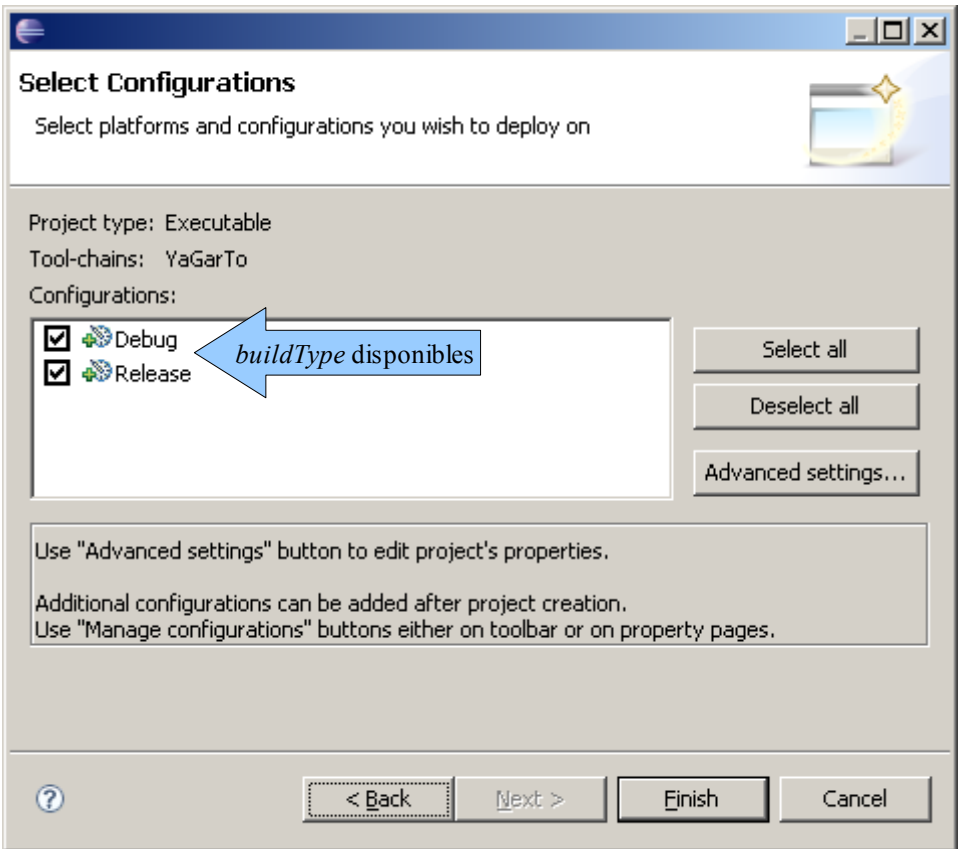
On peut voir à gauche les trois rubriques principales représentant les *buildArtefactType*. On aperçoit les deux qui nous intéressent à savoir les *Executable* ainsi que les *Static Librarie*. Dans ces deux rubriques les templates mis à disposition apparaissent. On constate qu'en dessous des templates de base apparaît notre template à savoir *Embedded ARMEBS3*.

Sur la partie de droite on retrouve la *toolchain* disponible pour notre template.

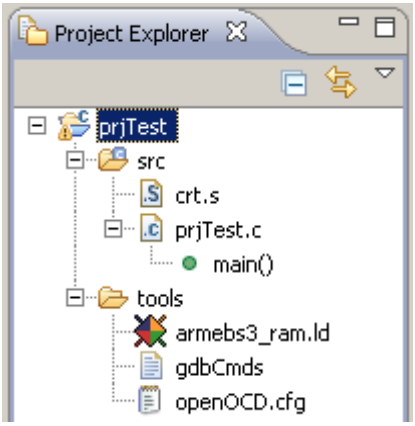
Création d'un exécutable



Après avoir sélectionné le type de projet désiré et la *toolchain* à utiliser (s'il y en a plusieurs) il faut cliquer sur le bouton *Next*. La dernière fenêtre apparaît. Celle-ci regroupe les *buildType* que nous avons déclarés dans les extensions. Par défaut les deux sont sélectionnés, il est cependant possible d'en choisir qu'un (*debug* par exemple) si on veut simplement faire un test.

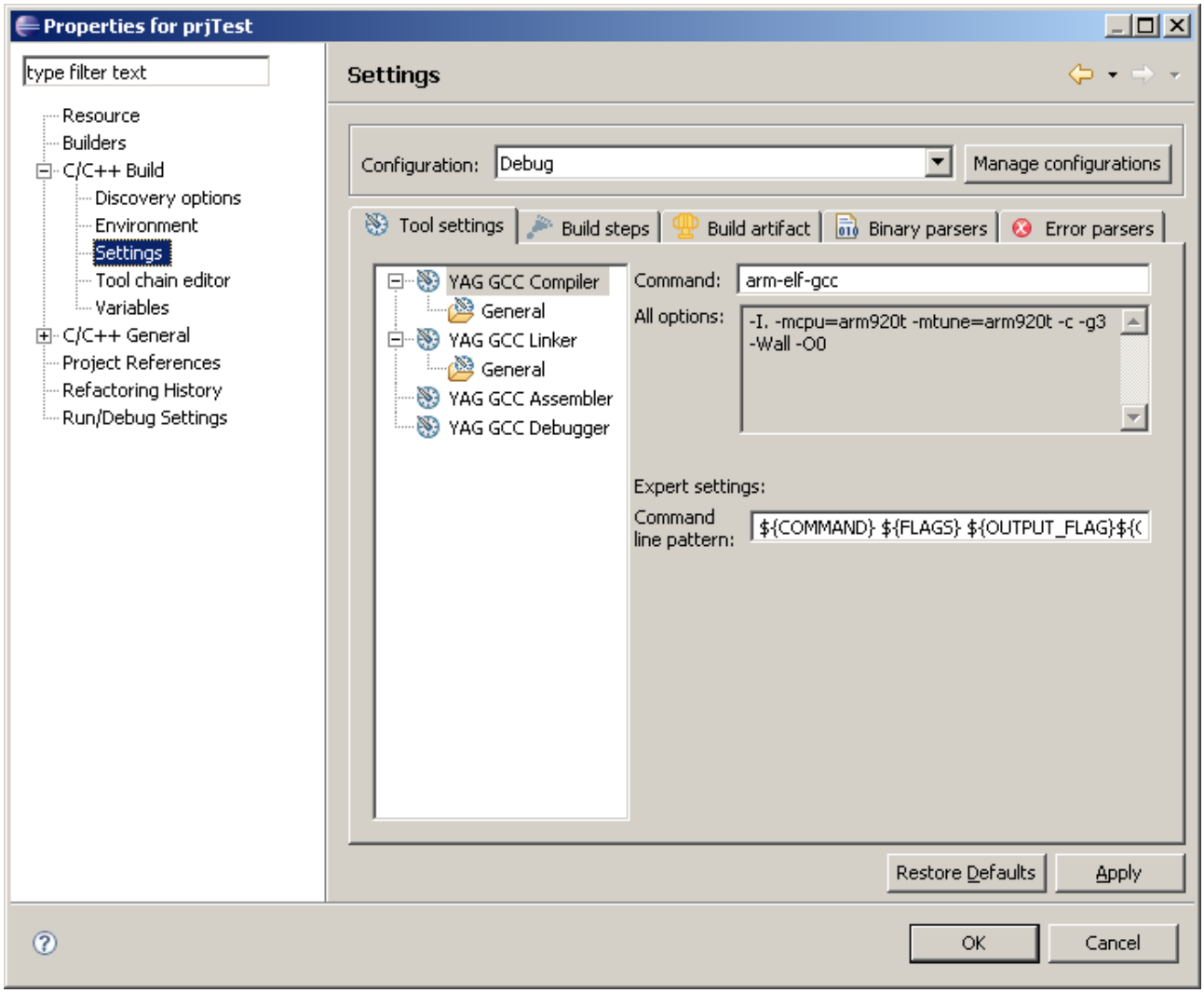


Pour terminer la configuration du projet *C* il faut cliquer sur le bouton *Finish*. A ce moment là, le plugin *Embedded HEVS* va exécuter le fichier *template.xml* celui-ci va réaliser les travaux suivants :

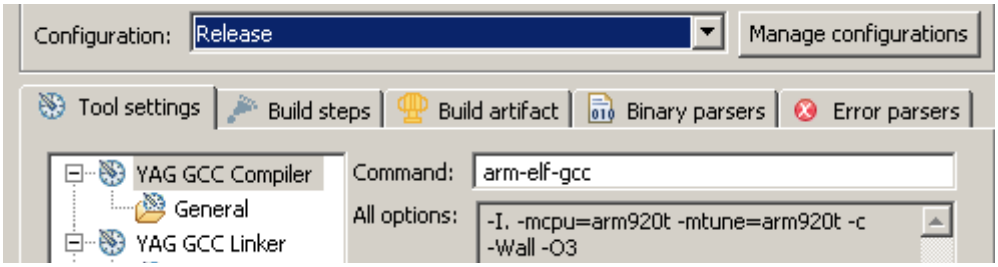


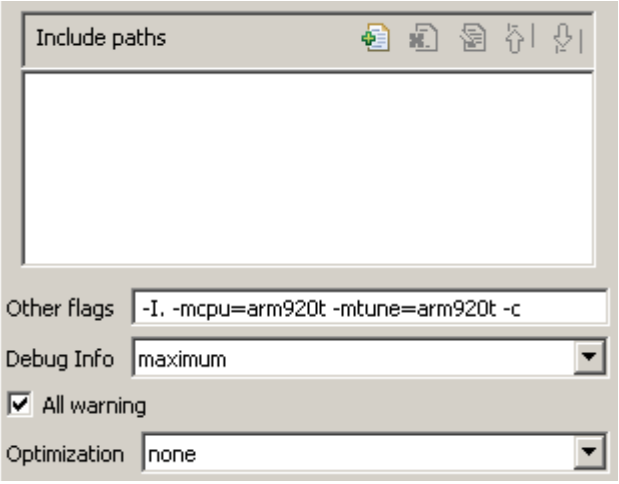
- création du dossier *src* contenant les fichiers sources
- ajout du fichier par défaut contenant le *main*
- copie du fichier *crt.s*
- création du dossier *tools*
- copie des trois fichiers nécessaires à la compilation et au debuggage d'un programme *.elf* à savoir les fichiers de lien, de commandes *gdb* et de configuration d'*openOCD*

Maintenant que les fichiers de base sont correctement chargés nous allons éditer les *settings* du projet. Pour ce faire il faut faire un clique droit sur le projet et sélectionner *properties*. Dans la rubrique *C/C++ Build* la section *settings* regroupe les informations relatives à la *toolchain*. On aperçoit la configuration courante (*debug* ou *release)* les outils à dispositions *(compiler, linker, assembler* et *debugger)* ainsi que la commande et les flags du compilateur.



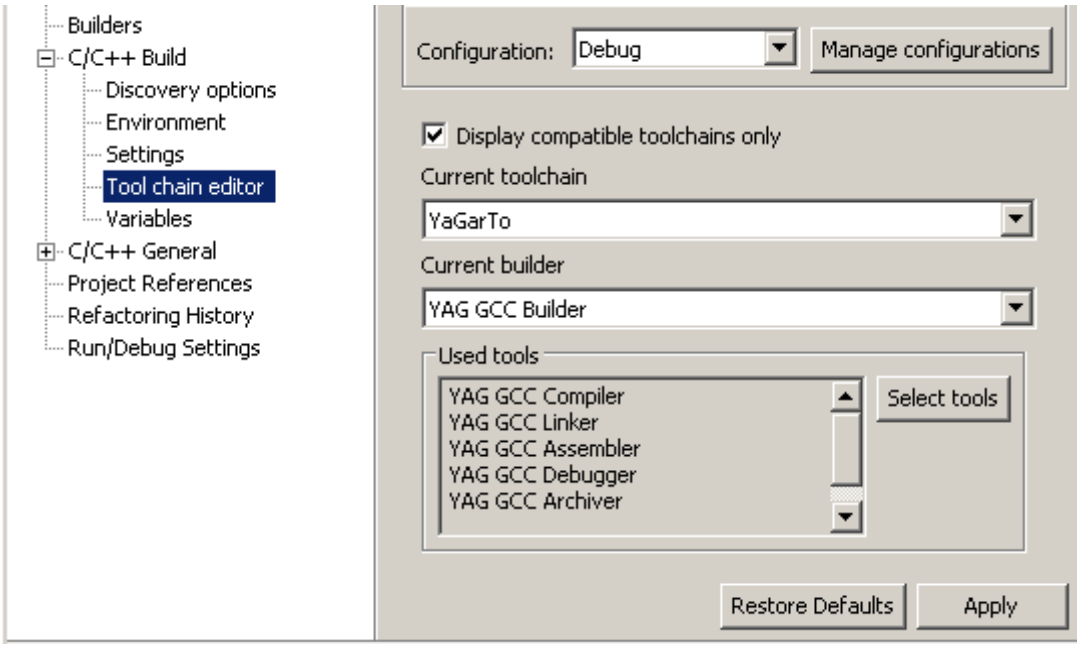
Comme expliqué plus haut, le fait de choisir la configuration *debug* ou *release* à comme effet de modifier les flags du compilateur :



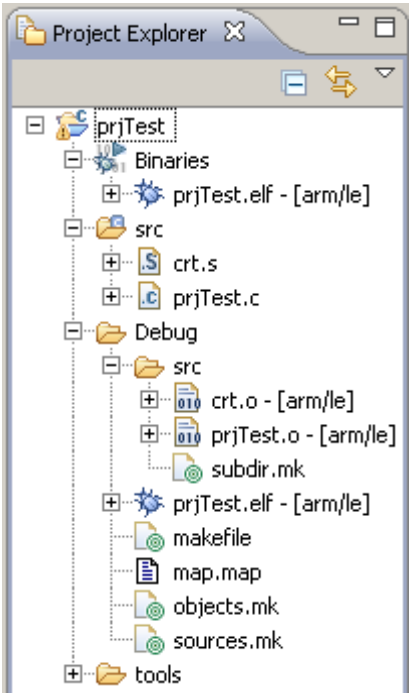


Lorsqu'on sélectionne la catégorie *General* du compilateur on constate que les options déclarées dans les extensions apparaissent. Tout d'abord, il est possible d'ajouter des *path* contenant des fichiers qui ne sont pas présents dans le projet *C*. Les flags par défaut sont bien chargés. De plus, les deux *combobox* contenant les flags relatifs au debug et à l'optimisation sont bien présentes. Une dernière option a été ajoutée, à savoir la coche *All warning* qui ajoute le flag *-Wall* à la commande du compilateur, ce flag permet de voir tous les *warnings* qui apparaissent lors de la compilation.

Après avoir parcouru les *settings* du projet il est possible de survoler la rubrique *Tool chain editor* qui permet de visualiser la *toolchain* utilisée et de lister les autres *toolchains* disponibles. On constate que le *builder* déclaré dans les extensions ainsi que touts les *tools* sont regroupés ici. Bien entendu tous les outils sont listés y compris l'archiveur même si celui-ci n'est pas utilisé pour un projet de type « Exécutable ».



Nous avons pu vérifier que les propriétés du projet *C* sont correctes. On peut donc affirmer que les extensions déclarées font bien leur travail. Voyons maintenant comment l'exécutable est construit. Pour ce faire il faut sélectionner le dossier du projet et cliquer sur *Build Project* dans le menu *Project*. A noter que si la coche *Build Automatically* de ce même menu est sélectionné le projet se construit tout seul à chaque modification de fichier source.



Si aucune erreur apparaît, l'exécutable est généré *(prjTest.elf)* un dossier portant le même nom que la configuration est créé, celui-ci contient les objets compilés ainsi que le *makefile* généré automatiquement (le *makefile* est séparé dans plusieurs fichiers). A noter que l'exécutable est représenté par un petit *bug* car celui-ci regroupe les informations relatives au *debug*.
On aperçoit aussi le fichier *map.map* car les flags du linker indique à l'éditeur de lien de générer un fichier contenant l'aperçu du mapping de la mémoire.

Le *makefile*

Voyons maintenant les fichiers permettant au *makefile* de construire l'application.

- makefile : ce fichier est le makefile principal. Il contient la règle principale qui permet de construire l'application *.elf*. A noter que la variable *$(OBJS)* représente l'*outputType* du compilateur, celle-ci à été déclarée dans les extensions du compilateur. Elle contient la liste des objets.

```
# Tool invocations
prjTest.elf: $(OBJS) $(USER_OBJS)
	@echo 'Building target: $@'
	@echo 'Invoking: YAG GCC Linker'
	arm-elf-gcc -I. -T "D:\prjTest\tools\armebs3_ram.ld" -nostartfiles
	-Xlinker -M -Xlinker -Map=map.map -o"prjTest.elf" $(OBJS)
	$(USER_OBJS) $(LIBS)
	@echo 'Finished building target: $@'
	@echo ' '
```

- objects.mk : contient les librairies ajoutées par l'utilisateur
- src/subdir.mk : ce fichier contient les règles nécessaires à la construction des objets, pour les fichier *.c* et *.s*

```
C_SRCS += \
../src/prjTest.c
```

La variable *C_SRCS* contient les fichiers sources rédigés en *C*.

```
S_SRCS += \
../src/crt.s
```

La variable *S_SRCS* contient les fichiers sources rédigés en assembleur.

```
OBJS += \
./src/crt.o \
./src/prjTest.o
```

La variable *OBJS* contient les objets compilés.

```
src/%.o: ../src/%.s
	@echo 'Building file: $<'
	@echo 'Invoking: YAG GCC Assembler'
	arm-elf-as  -o"$@" "$<"
	@echo 'Finished building: $<'
	@echo ' '
```

Cette règle permet de compiler les fichiers sources rédigés en assembleur en invoquant la commande associée à savoir *arm-elf-as*.

```
src/%.o: ../src/%.c
	@echo 'Building file: $<'
	@echo 'Invoking: YAG GCC Compiler'
	arm-elf-gcc -I. -mcpu=arm920t
	            -mtune=arm920t -c
	            -g3 -Wall -O0 -o"$@" "$<"
	@echo 'Finished building: $<'
	@echo ' '
```

Cette règle permet de compiler les fichiers sources rédigés en *C* en invoquant *arm-elf-gcc* et en utilisant les flags que nous avons spécifiés plus haut.

La console

Le *makefile* est maintenant généré. Voyons ce qui se passe lorsque la commande *make* est invoquée. On constate que la règle permettant de construire l'exécutable est appelée. Cette dernière dépend des fichiers objets, leur compilation est donc invoquée. Pour finir, l'exécutable est linké on constate que les flags sont passés en paramètre et que l'exécutable est généré correctement.

```
**** Build of configuration Debug for project prjTest ****

make all
Building file: ../src/crt.s
Invoking: YAG GCC Assembler
arm-elf-as  -o"src/crt.o" "../src/crt.s"
Finished building: ../src/crt.s

Building file: ../src/prjTest.c
Invoking: YAG GCC Compiler
arm-elf-gcc -I. -mcpu=arm920t -mtune=arm920t -c -g3 -Wall -O0 -o"src/prjTest.o"
"../src/prjTest.c"
Finished building: ../src/prjTest.c

Building target: prjTest.elf
Invoking: YAG GCC Linker
arm-elf-gcc -I. -T "D:\prjTest\tools\armebs3_ram.ld" -nostartfiles -Xlinker -M
-Xlinker -Map=map.map -o"prjTest.elf"  ./src/crt.o ./src/prjTest.o
Finished building target: prjTest.elf
```

règle principale

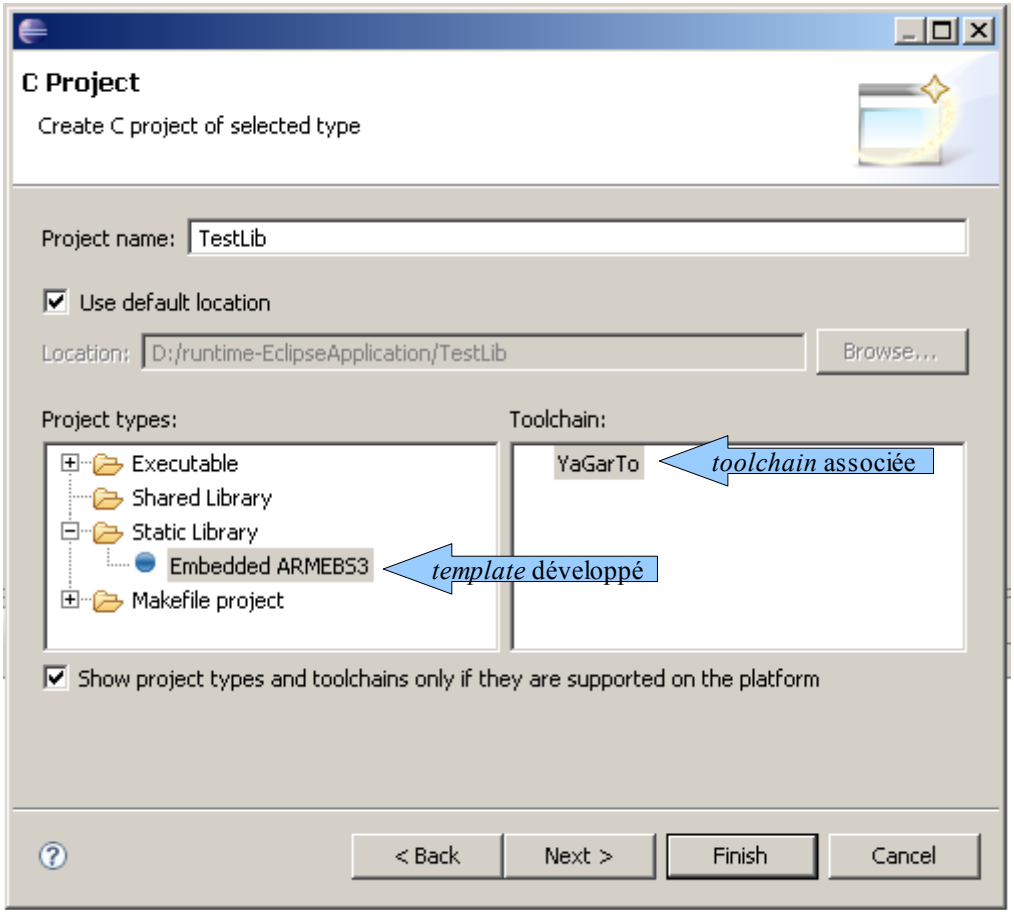
compilation

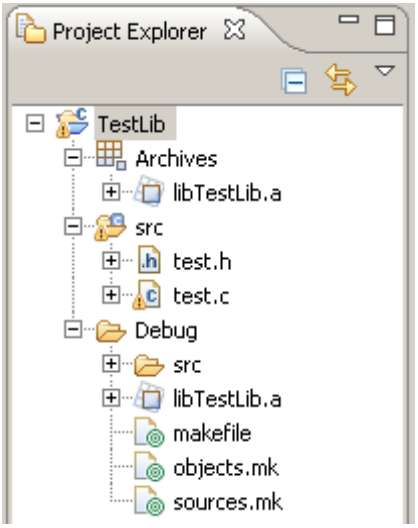
compilation

link de l'exe

Création d'une libraire

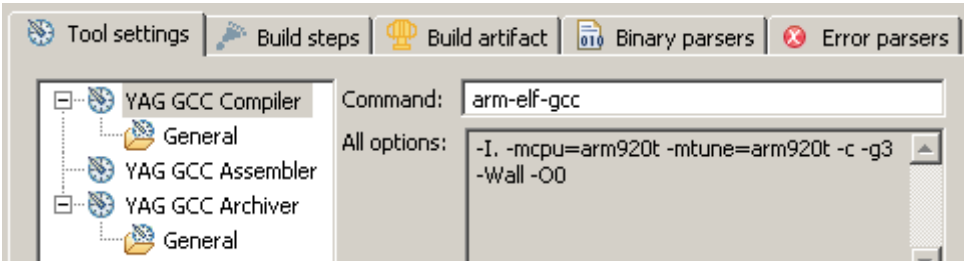
Comme pour l'exécutable la création d'une libraire s'effectue très rapidement





Après avoir cliqué sur *Finish* un projet vide est créé. Pour tester la construction de la librairie, le *.a*, nous avons créé un header *test.h* et un fichier source *test.c* contenant une petite fonction. Comme pour l'exécutable un dossier portant le nom de la configuration est créé, celui-ci contient le makefile séparé dans plusieurs fichiers et on constate que la librairie est correctement générée.

Puisque le projet est une librairie, la *toolchain* est composée d'un *compilateur*, d'un *assembleur* et d'un *archiver*. Nous allons vérifier que c'est bien le cas en affichant les settings du projet.



Tout se passe bien, il est bien entendu possible, comme pour l'exécutable, de choisir la configuration souhaitée, *debug* ou *release*. Pour que le test soit complet voyons encore l'output de la commande *make*.

```
**** Build of configuration Debug for project TestLib ****

make all
Building file: ../src/test.c
Invoking: YAG GCC Compiler
arm-elf-gcc -I. -mcpu=arm920t -mtune=arm920t -c -g3 -Wall -O0 -o"src/test.o"
"../src/test.c"
Finished building: ../src/test.c

Building target: libTestLib.a
Invoking: YAG GCC Archiver
arm-elf-ar -r "libTestLib.a"  ./src/test.o
c:\yagarto\bin\arm-elf-ar.exe: creating libTestLib.a
Finished building target: libTestLib.a
```

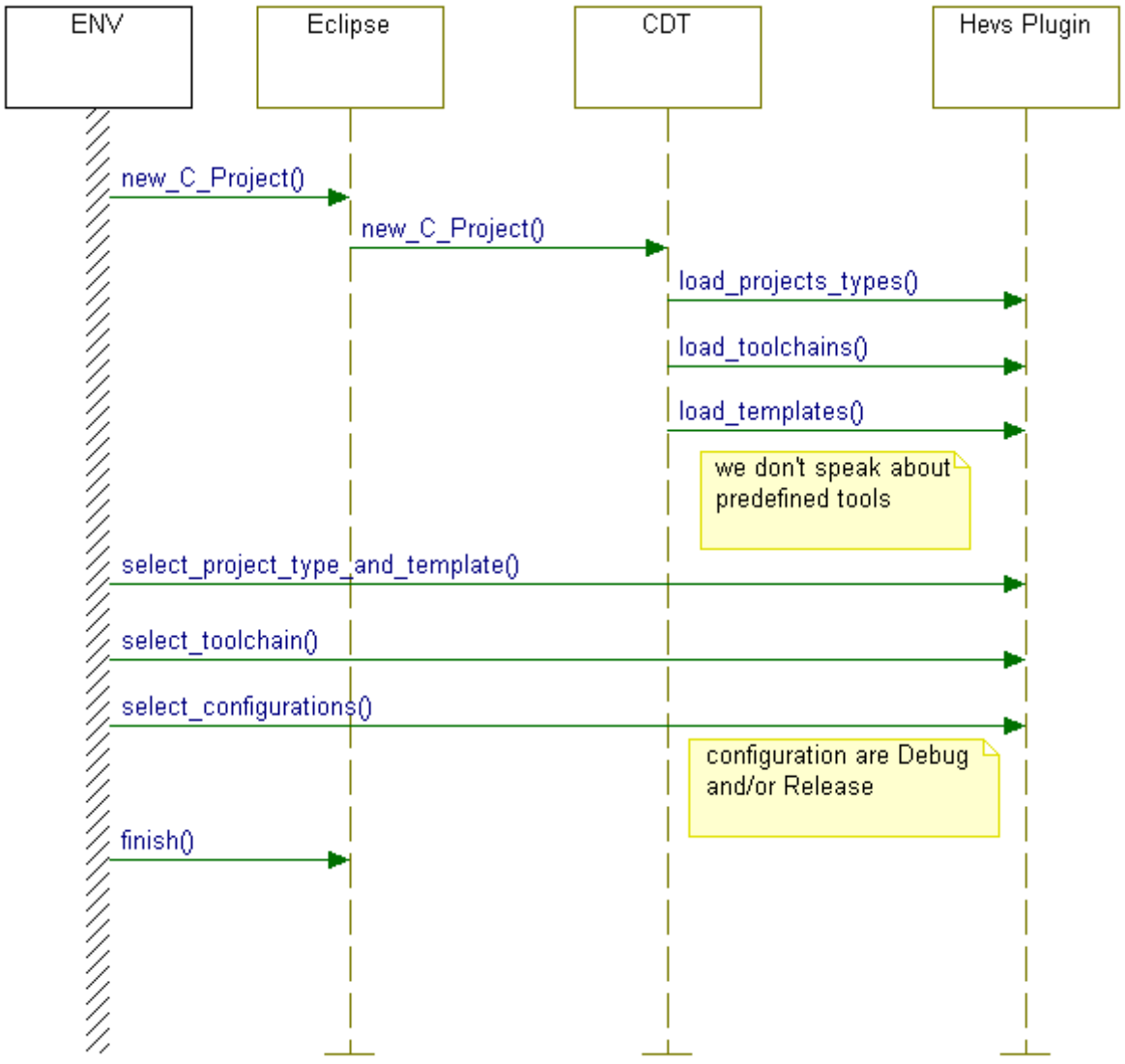
règle principale

compilation

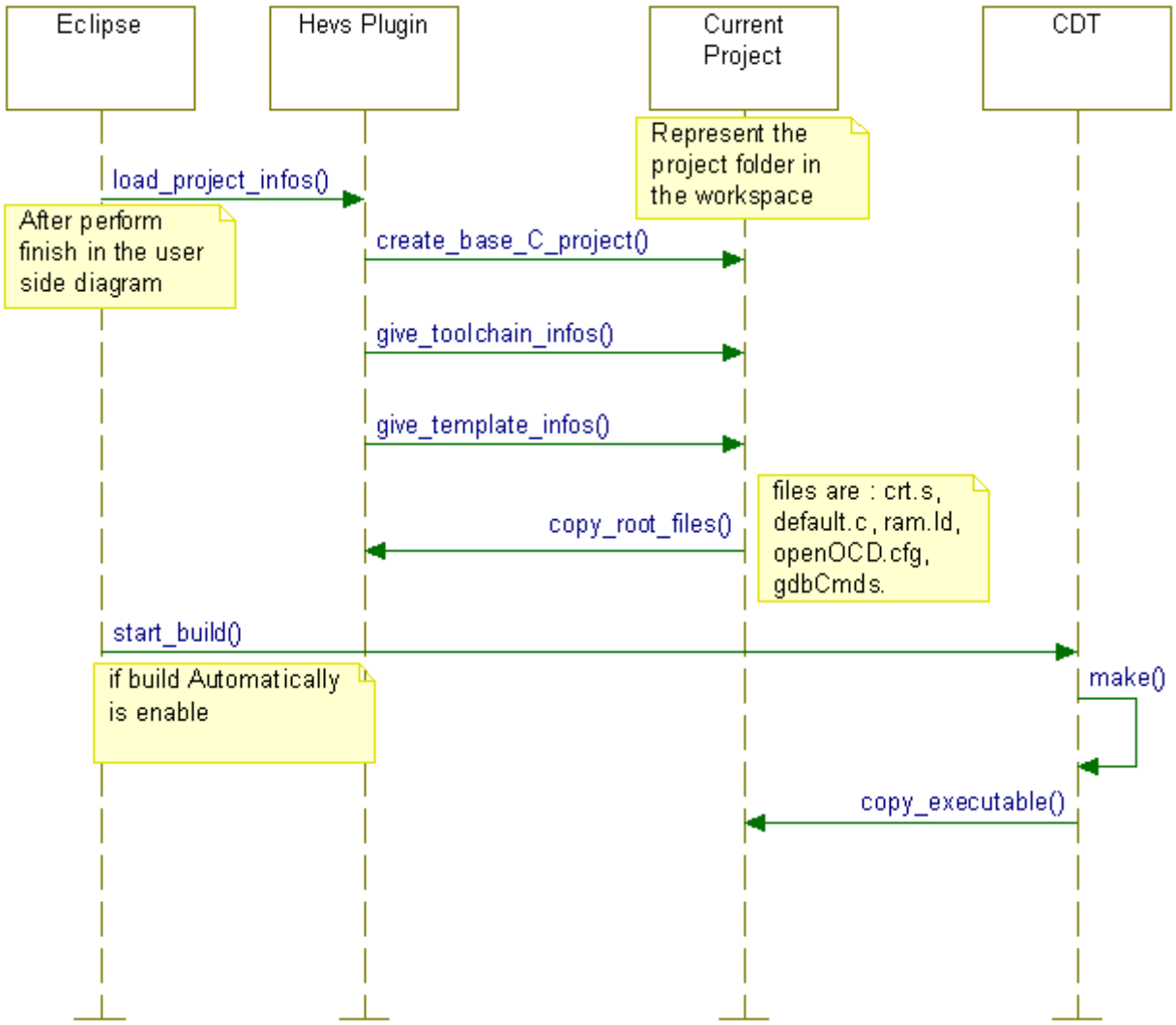
archivage des objets

Une aide supplémentaire concernant les makefiles est ajoutée au CD-Rom livré avec ce rapport

Configuration d'un nouveau projet C (résumé)



Construction du poject C dans le workspace (résumé)



Les plugins Zylin

Zylin est une entreprise norvégienne oeuvrant notamment dans les technologies embarquées. Elle met à disposition un PCB programmable et propose aussi deux plugins Eclipse permettant de programmer cette carte et de la debugger. Bien entendu ces plugins sont opensources.

Zylin Embedded CDT

Le plugin CDT 4.0 a un excellent support permettant de debugger avec *gdb*. Cependant il persiste quelques classes et parties de code qui posent problème lors du debuggage d'applications embarquées. Ce plugin effectue quelques modifications du plugin CDT 4.0. Il faut simplement l'installer en parallèle du plugin CDT 4.0

Zylin CDT 4.0

Ce plugin permet de debugger une application embarquée en natif et via cygwin. Pour les extensions *Embedded debugging* j'ai principalement repris les classes fournies par ce plugin. Par la suite, je vais préciser les modifications apportées à ces classes et parler plus en profondeur des classes ajoutées permettant d'automatiser l'utilisation d'*OpenOCD*.

Les *extensions « Embedded debugging »*

Après le développement des extensions permettant la gestion de nouveaux projets, nous allons poursuivre en ajoutant les extensions relatives au téléchargement et au debuggage du code sur la cible depuis l'IDE.

org.eclipse.core.runtime.preferences

Cette extension permet de préciser quelle classe déclare les préférences par défaut qui seront utilisées par le plugin. Le plugin *Embedded Hevs* contient à ce jour deux préférences. La première nous donne le chemin relatif du fichier de commandes *gdb*. La deuxième est plus intéressante, elle précise le chemin du programme permettant d'exécuter *OpenOCD*. La méthode *initializeDefaultPreferences* de la classe *PrefInitializer* déclare ces préférences :

```
public void initializeDefaultPreferences()
{
   IPreferenceStore store = LaunchPlugin.getDefault().getPreferenceStore();
   store.setDefault(PrefConstants.P_DEBUGGER_INIT, LaunchPlugin
                .getResourceString(PrefConstants.P_RES_DEBUGGER_INIT));
   store.setDefault(PrefConstants.P_OPENOCD_CMD, LaunchPlugin
                .getResourceString(PrefConstants.P_RES_OPENOCD_CMD));
}
```

Toujours dans le package *hevs.embedded.preferences* on retrouve la classe *PrefConstants* regroupant les constantes relatives aux préférences décrites ci-dessus. Il faut savoir qu'une constante est représentée par un identifiant et par une valeur. Voici comment cette classe déclare ces constantes :

```
public class PrefConstants {

	public static final String P_DEBUGGER_INIT = "debuggerInit";
	public static final String P_OPENOCD_CMD = "openocdcmd";

	public static final String P_RES_OPENOCD_CMD = "C:/openocd/bin" +
	                                               "/openocd-ftd2xx.exe";
	public static final String P_RES_DEBUGGER_INIT = "tools/gdbCmds" ;

}
```

Par exemple, l'identifiant du liens vers *OpenOCD* est représenté par la constante *P_OPENOCD_CMD* et la valeur associée à cet identifiant est contenue dans la constante *P_RES_OPENOCD_CMD*.

Utiliser ces préférences

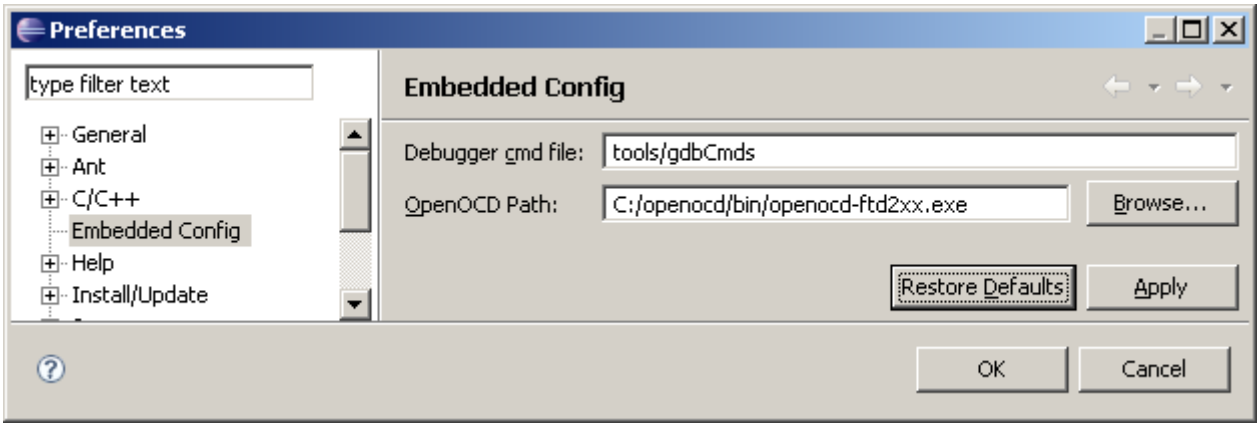
Pour accéder à ces préférences il va falloir utiliser la classe de base du plugin, à savoir *LaunchPlugin* qui permet d'accéder à plusieurs outils. Voici comment procéder.

```
LaunchPlugin.getDefault().getPluginPreferences().getString(getValue(PrefConst
ants.P_OPENOCD_CMD));
```

Cette instruction va permettre de récupérer la chaîne de caractère contenu dans la constante *P_RES_OPENOCD_CMD*.

org.eclipse.ui.preferencePages

Les préférences sont malheureusement enregistrées dans des constantes. Il serait pourtant intéressant de pouvoir les modifier dès lors que le chemin ver *OpenOCD* change par exemple. C'est pourquoi une extension supplémentaire a été ajoutée. Cette dernière gère une interface graphique permettant d'afficher et de modifier ces constantes. La classe *PreferencePage* contenue elle aussi dans le package *hevs.embedded.preferences* gère l'interface graphique. Pour accéder à ces informations il faut sélectionner le menu *Window* puis *Preferences*... Dans la rubrique *Embedded Config* il est possible d'accéder aux informations décrites plus haut. A l'aide du *bouton Restore Default* il est toujours possible de resélectionner les informations par défaut contenues dans les constantes de base.



Gestion de la configuration d'un projet C

Les classes abordées ci-dessous utilisent un nouvel aspect du développement de plugin à savoir l'enregistrement de la configuration relative au projet. Par exemple, si l'utilisateur veut modifier le nom du debugger ou du fichier de commandes il faut pouvoir enregistrer ces modifications dans la configuration du projet *C*. Une interface est déclarée regroupant, comme pour les préférences des chaînes de caractères représentant l'identifiant de la valeur à enregistrer dans la configuration. L'interface *LaunchConfigurationConstants* contient ces informations. Par exemple, l'identifiant du path d'*OpenOCD* est représenté par *ATTR_OPENOCD_PATH*

```
public interface LaunchConfigurationConstants {
   static final String LAUNCH_ID = "hevs.embedded";
   String ATTR_DEBUGGER_COMMANDS_INIT =LAUNCH_ID + ".debugger_init_commands";
   String ATTR_DEBUGGER_COMMANDS_RUN =  LAUNCH_ID + ".debugger_run_commands";
   String ATTR_OPENOCD_PATH = LAUNCH_ID + ".openocd_path";
   String ATTR_OPENOCD_CONFIG_FILE = LAUNCH_ID + ".openocd_config_file";
}
```

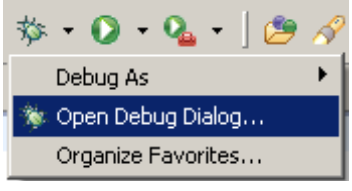
Gestion d'une interface graphique typique

Les classes liées aux interfaces graphiques permettant de configurer le lancement du debuggage sont composées entre autre de trois méthodes bien définies et appelées à des moments précis. Ces méthodes doivent être redéfinies car ces classes héritent toutes d'*AbstractLaunch-ConfigurationTab*. Les méthode à redéfinir sont les suivantes :

- *setDefaults* : cette méthode est appelée seulement une fois, lorsque l'on créé une nouvelle configuration de lancement. Elle permet de remplir les champs avec des valeurs de base. Cette méthode permet typiquement de charger les valeurs contenues dans les constantes décrites plus haut.
- *initializeFrom* : cette méthode est appelée lors de l'initialisation de l'interface graphique. Elle permet de remplir les champs en récupérant les données enregistrées dans la configuration.
- *performApply* : cette méthode récupère les données inscrites dans les champs et les enregistre dans la configuration. A noter que cette méthode est appelée à chaque fois qu'un champ est modifié.

org.eclipse.debug.ui.launchConfigurationTabGroups

Cette extension regroupe un certain nombre d'onglets permettant de configurer un type de lancement pour un exécutable. Par lancement on entend la manière dont le programme est exécuté: en local ou en embarqué par exemple. La nouvelle configuration est accessible via le menu debug.



Une classe est associée à cette extension elle permet de créer les onglets regroupant la configuration du lancement. Voici le code le plus représentatif de cette classe.

```
public void createTabs(ILaunchConfigurationDialog dialog, String mode) {
ILaunchConfigurationTab[] tabs = new ILaunchConfigurationTab[] {
            new MainTab(),
            new EmbeddedDebuggerTab(false),
            new OpenOCDTab(),
            new CommandTab(),
            new SourceLookupTab(),
            new CommonTab()
      };
```

Les deux classes les plus importantes présentent ici sont la classe *OpenOCDTab et EmbeddedDebuggerTab* contenant la classe Embedde*dGDBDebuggerP*age

EmbeddedGDBDebuggerPage

Cette classe représente la partie inférieur de l'onglet *EmbeddedDebuggerTab.* Elle regroupe les informations nécessaires au debug. A savoir le nom du debugger (l'information est récupérée depuis la *toolchain* du Projet *C*) ainsi que le nom du fichier contenant les commandes à envoyer à *gdb* (l'information est récupérée depuis les préférences). La récupération de ces informations se fait via les trois méthodes décrites ci-dessus.

- *setDefaults* : après avoir créé une nouvelle configuration de lancement cette méthode va récupérer dans les préférences le nom du fichier contenant les commandes à envoyer à *gdb* ainsi que le nom du debugger via la méthode *getDebuggerName.* Ces informations sont ensuite enregistrées dans la configuration via la méthode *setAttribute(id, string).*

```
public void setDefaults(ILaunchConfigurationWorkingCopy configuration)
{
    String gdbCmdFile = getValue(PrefConstants.P_DEBUGGER_INIT);

    configuration.setAttribute(
            IMILaunchConfigurationConstants.ATTR_DEBUG_NAME, getDebuggerName());
    configuration.setAttribute(
            IMILaunchConfigurationConstants.ATTR_GDB_INIT, gdbCmdFile);
}
```

GetDebuggerName

Cette méthode parcoure les outils de la *toolchain* et récupère la commande associée à l'outil possédant le string «debugger » dans son nom. Pour plus d'informations concernant cette méthode veuillez vous référer au code.

- *initializeFrom* : cette méthode va récupérer les valeurs contenues dans la configuration et les copier dans les champs associés.
- *performApply* : cette méthode va récupérer les valeurs des champs et mettre à jour la configuration si des modifications on été apportées.

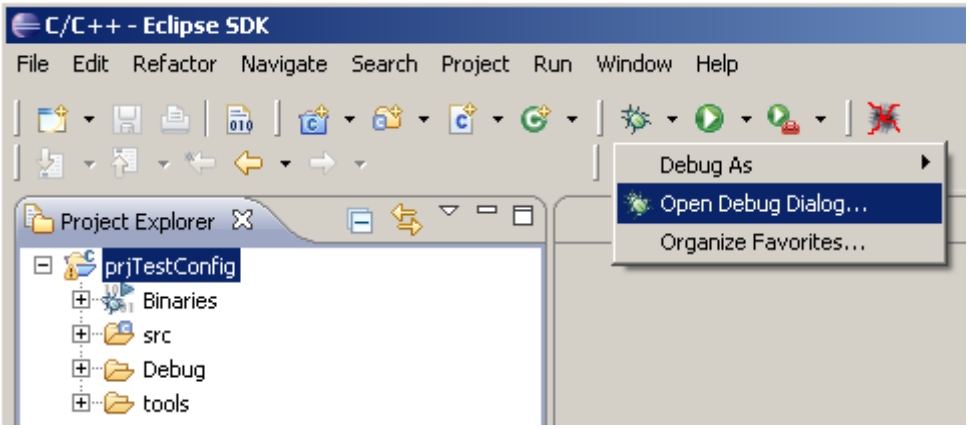
OpenOCDTab

Cette classe possède exactement le même comportement que la classe décrite ci-dessus. Elle possède aussi les trois méthodes caractéristiques. Le rôle de cette classe est de récupérer le path d'*OpenOCD* ainsi que le chemin complet du fichier de configuration contenu dans le dossier tool du projet courant. Pour bien comprendre son fonctionnement nous allons voir de plus près le code contenu dans la méthode *performApply* :

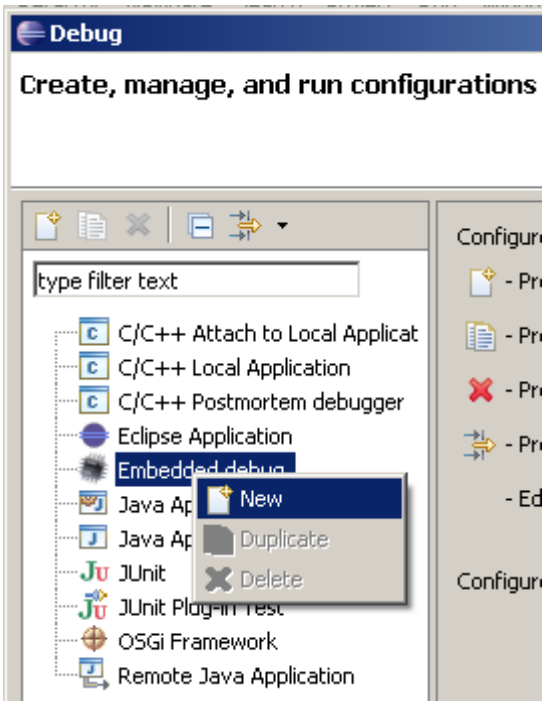
```
public void performApply(ILaunchConfigurationWorkingCopy configuration)
{
   String configPath = getAttributeValueFrom(txtConfigFile) ;

      //the path is unknown
   if (configPath.toLowerCase().indexOf("unknown") != -1)
      configPath = getOpenOCDFilePath() ;


   configuration.setAttribute(
                        LaunchConfigurationConstants.ATTR_OPENOCD_CONFIG_FILE,
                        configPath);
   configuration.setAttribute(
                        LaunchConfigurationConstants.ATTR_OPENOCD_PATH,
                        getAttributeValueFrom(txtOpenOcdPath));
}
```

La méthode *getAttributeValueFrom* permet de récupérer une copie du champ *txtConfigFile* si la valeur du champs est inconnue, « unknown », on récupère le path via la méthode *getOpenOCDFilePath*. La chaîne de caractères « unknown » est attribuée au champs si aucun projet n'est sélectionné lors de la création de la configuration de lancement. Après avoir récupéré correctement les données ces dernières sont enregistrées dans la configuration.

Test de la configuration pour lancer le debug



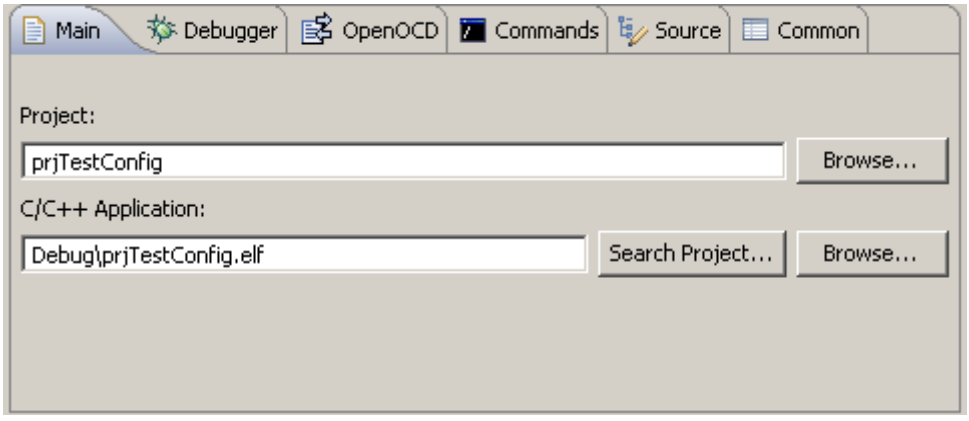
Dans un premier temps il faut créer un projet *C* en sélectionnant le template *Embedded ARMEBS3*. L'exécutable doit bien entendu être compilé et linké correctement. Pour créer une nouvelle configuration de debug il faut sélectionner le projet correspondant et cliquer sur la flèche de l'icône correspondant à un bug.



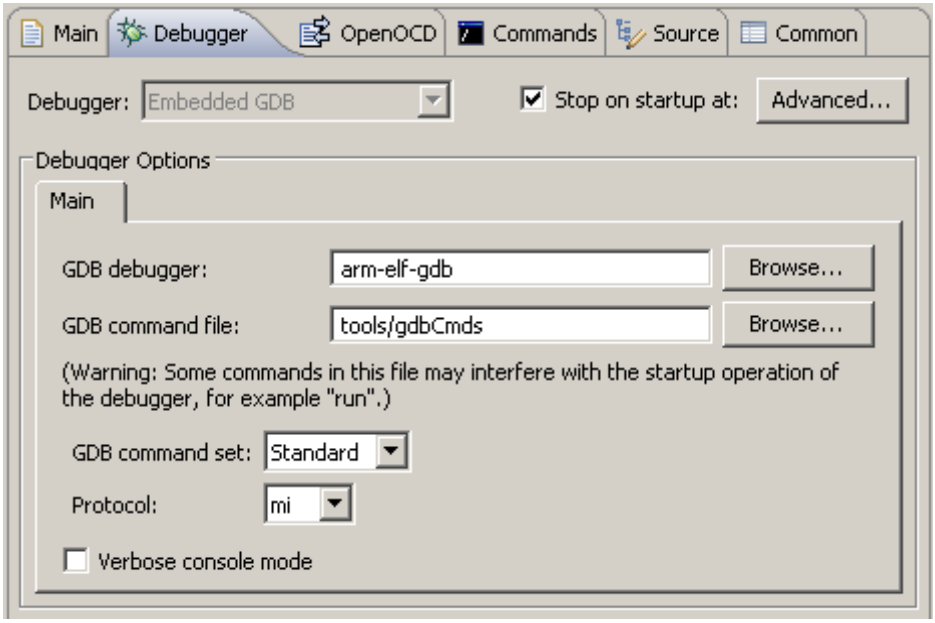
La fenêtre principale permettant de debugger des programmes apparaît alors. On aperçoit dans la liste de gauche les configurations disponibles. Notre configuration s'appelle *Embedded debug* et est représentée par un petit processeur. Pour créer la configuration il faut effectuer un clique droit sur la rubrique et sélectioner *New*

Apparaît alors la fenêtre composée des différents onglets déclarés dans la classe *LaunchConfigurationTabGroup*. Voici un aperçu des onglets relatifs au plugin *Embedded Hevs* mis à disposition de l'utilisateur.

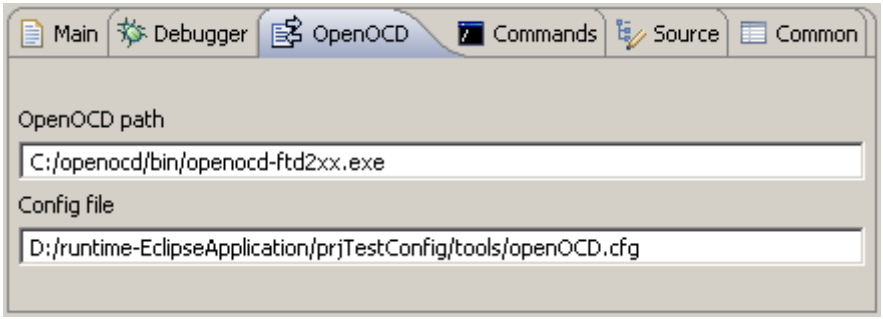
- Main : cet onglet affiche le projet courant ainsi que l'exécutable qui sera debuggé. Il est bien entendu possible de les rechercher si le projet n'était pas sélectionné lors de la création de la configuration.



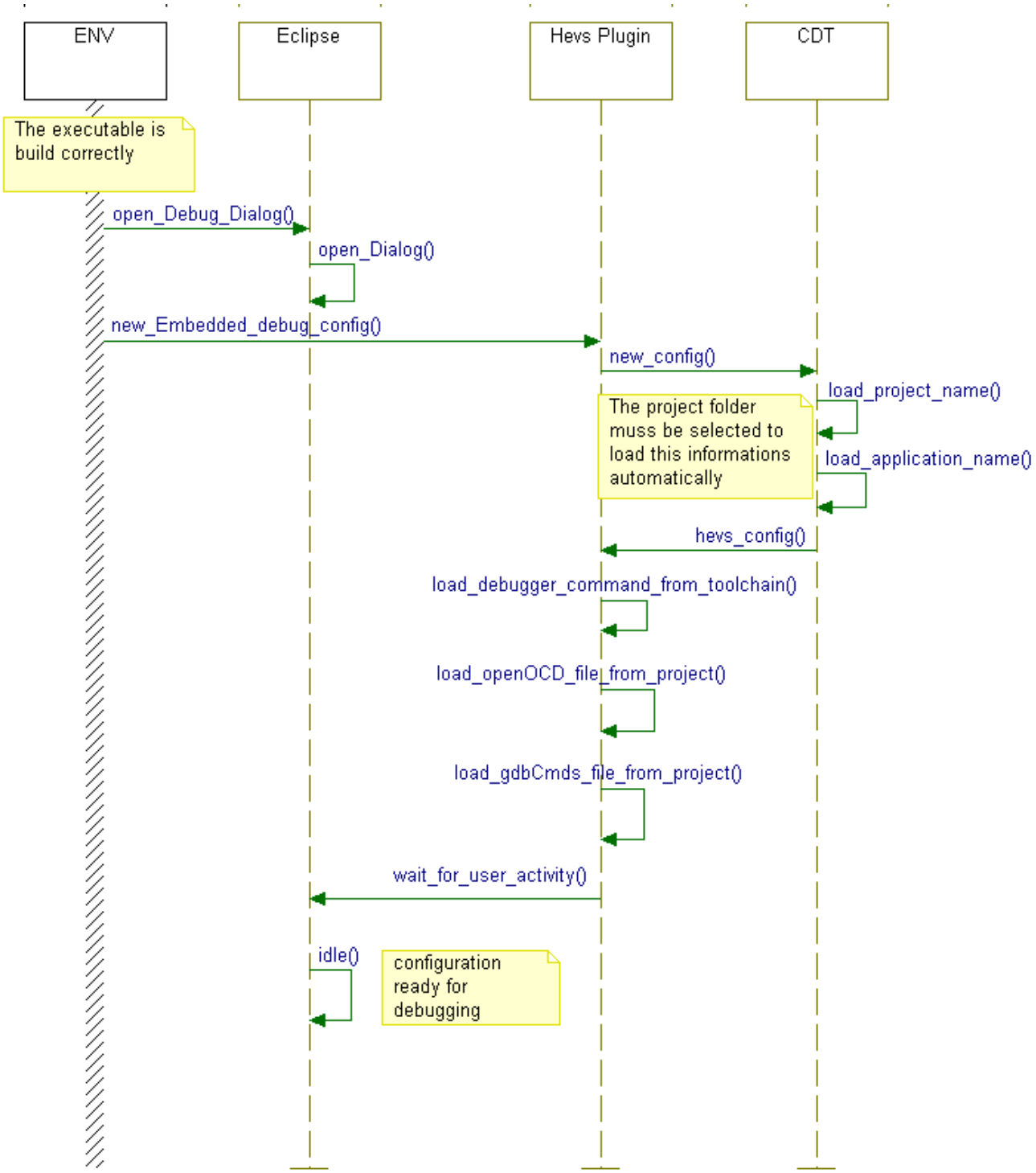
- Debugger : ici sont rassemblées les informations concernant le debugger utilisé ainsi que le path relatif du fichier de commandes *gdb*. On constate donc que la recherche de la commande *arm-elf-gdb* ainsi que l'accès au préférences fonctionnent correctement.



- OpenOCD : cet onglet regroupe les informations concernant *OpenOCD*. A savoir le path de l'exécutable sur le disque (récupéré depuis les préférences) ainsi que le path complet du fichier de configuration d'*OpenOCD*. Ce fichier est propre au projet courant.



Création d'une nouvelle configuration (résumé)



Gestion d'OpenOCD

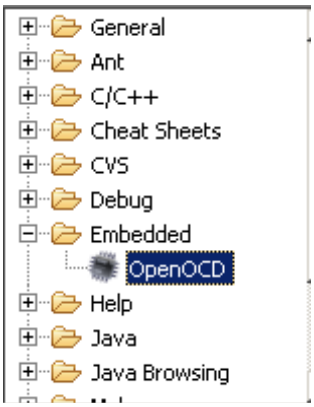
Pour pouvoir exécuter le debug sur la carte il faut bien entendu lancer l'exécutable *openocd-ftd2xx.exe*. D'une part, il faut que le plugin gère lui-même le démarrage et l'arrêt d'*OpenOCD* et d'autre part il faut récupérer l'output du programme pour l'afficher à l'utilisateur.

Une extension permettant d'ajouter une console doit être développée ainsi qu'une classe manageant le démarrage et l'arrêt d'*OpenOCD*. Cette classe possède entre autre son propre *thread* permettant de récupérer les informations affichées par *OpenOCD*.

org.eclipse.ui.views

Cette extension ajoute une catégorie dans le menu permettant de sélectionner une vue spécifique au plugin *Embedded Hevs*. Cette vue va contenir une console graphique permettant d'afficher les informations récupérées depuis l'exécutable *openocd-ftd2xx.exe*. Deux extensions seront nécessaires, la première ajoute une catégorie (Embedded) et la deuxième représentera la vue supplémentaire (OpenOCD). On constate qu'une classe représentant la vue est ajoutée ainsi que l'identifiant de la catégorie Embedded.

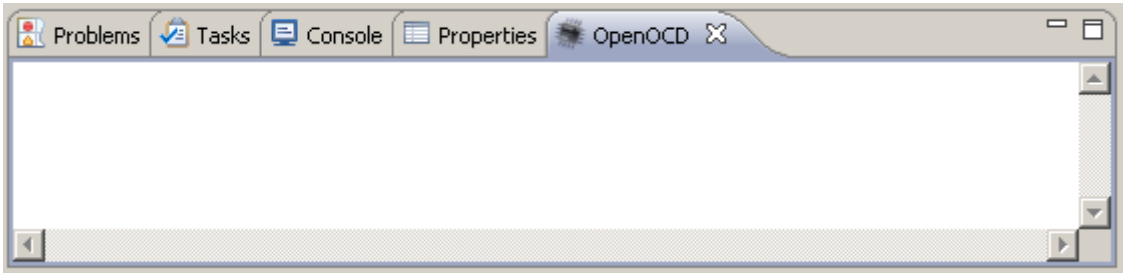
La classe associée contient les objets *TextConsoleViewer* et *MessageConsole*. Ces deux objets vont permettre d'ajouter une console à la vue. Il suffira ensuite d'appeler les méthodes correspondantes pour afficher des informations dans la console. Pour accéder à la nouvelle vue créée il faut cliquer sur le menu *Windows – Show View – Other...*



Dans la fenêtre *Show View*, on constate que la catégorie *Embedded* apparaît dans la liste des catégories mises à disposition. Cette catégorie contient bien la nouvelle vue *OpenOCD*.

Pour l'afficher il faut la sélectionner puis cliquer sur *OK*.

On constate alors qu'une nouvelle vue apparaît au niveau de la Console standard. Comme prévu cette vue porte la petite icône représentant un processeur. Cette classe est maintenant un outil disponible pour l'affichage d'informations.



La classe ManageOpenOCD

C'est la classe principale permettant de gérer *OpenOCD*. Voici les informations primordiales concernant cette classe :

- Contient un objet de type *Process* permettant d'exécuter et d'arrêter un processus externe.
- Déclare et utilise la classe *ManagePrint* qui accède au stream d'*openOCD* et utilise la console décrite plus haut pour afficher les informations reçues. Cette classe est un *thread* indépendant.
- Déclare le *thread KillOpenOCD* qui s'exécute automatiquement lors de la fermeture d'Eclipse. Ce *thread* détruit le processus *openocd-ftd2xx.exe* et arrête l'exécution du *thread ManagePrint.*

Les méthodes

- Le constructeur : met à disposition de l'extérieur une variable statique pointant sur lui-même. Il ajoute aussi le *thread KillOpenOCD* au *Runtime* principal.

```
public ManageOpenOCD ()
{
	if (openOCD == null)
	{
		openOCD = this ;

		//we had the thread to the Runtime
		Runtime.getRuntime().addShutdownHook(new KillOpenOCD()) ;

		System.out.println("Manager create") ;
	}
}
```

La notion de « pointeur sur lui-même » ou « pointeur statique » sera expliquée plus en détail par la suite.

- setOpenOCDInfos : Cette méthode permet d'initialiser les deux variables contenant le chemin de l'exécutable *openocd-ftd2xx.exe* ainsi que le path du fichier de configuration. Cette méthode doit absolument être appelée avant la méthode *restart()*.
- restart : Dans un premier temps le processus et le thread sont tués puis ces derniers sont reconstruits puis exécutés.

```
public void restart ()
{
    if (openOCD != null && openOCDConfigFile != null)
    {
      try
      {
        kill () ;

        ProcessBuilder builder = new ProcessBuilder(openOCDCmd, "-f",
                                     openOCDConfigFile) ;
        //C:/openocd-2007re204/bin/openocd-ftd2xx -f
                                     D:/at91r40008_jtagkeyARMEBS.cfg

        exe = builder.start() ;
        threadPrinter = new ManagePrint() ;
        threadPrinter.start() ;
       }
       catch (Exception e) {System.out.println("error") ;}
     }
}
```

- kill : permet de tuer le processus et le *thread*. A noter que cette méthode peut prendre un peu de temps car on attend que le processus soit bel et bien mort.

  Le code de gauche tue le processus et le code de droite arrête le *thread*

```
if (exe != null)
{
        exe.destroy() ;

    try
    {
        //wait for exe death
        exe.waitFor() ;
    }
    catch (InterruptedException e) {}

    exe = null ;
}
```

```
if (threadPrinter != null)
{
      threadPrinter.halt() ;

      //wait for the halt of the thread
      while (threadPrinter.isAlive()){}

      threadPrinter = null ;
}
```

La classe *ManagePrint*

Cette classe hérite de *java.lang.Thread* elle est déclarée à l'intérieur même de la classe *ManageOpenOCD,* ainsi les objets déclarés dans cette dernière sont accessibles directement. *ManagePrint* contient la méthode *run* qui est appelée lorsque le *thread* est exécuté. Pour commencer, cette méthode récupère le stream d'*OpenOCD*. A noter que les informations sont envoyées au *standard error stream* et non pas au *standard output stream*.

```
public void run ()
{
   boolean fin = false;
   String currentLine = null ;
   infoReader = new BufferedReader(new InputStreamReader(exe.getErrorStream()));
   //...
```

Ensuite une boucle est mise en place. Cette dernière récupère les informations reçues depuis le processus *openocd-ftd2xx.exe* et les affiche via la console ajoutée précédemment. La méthode *readLine()* du *BufferedReader* est bloquante, c'est pourquoi l'utilisation d'un *thread* indépendant est impératif.

```
    while (!fin)
    {
        //the infoReader.readLine() method is blocking the Thread
        //we must destroy the openOCD process before halt this Thread
        while((currentLine = infoReader.readLine()) != null)
            OpenOCDConsole.getDefault().print(currentLine) ;
        //...
```

On peu constater que la boucle principale se termine dès lors que la variable *fin* passe à *true.* La fin de cette boucle va permettre de terminer le *thread*. Nous savons cependant que la deuxième boucle permettant de lire le stream est aussi bloquante. Il est donc impératif de tuer le processus *openocd-ftd2xx.exe* avant de modifier la valeur du flag *fin.* La méthode kill effectue ces opérations dans le bon ordre. La méthode *halt* est synchronisée. L'exécution de son contenu sera alors verrouillé et aucun autre *thread* ne pourra l'appelé en même temps :

```
public synchronized void halt()
{
    this.stopThread = true ;
    //close the stream from openocd
    try{infoReader.close();}catch (Exception e){}
}
```

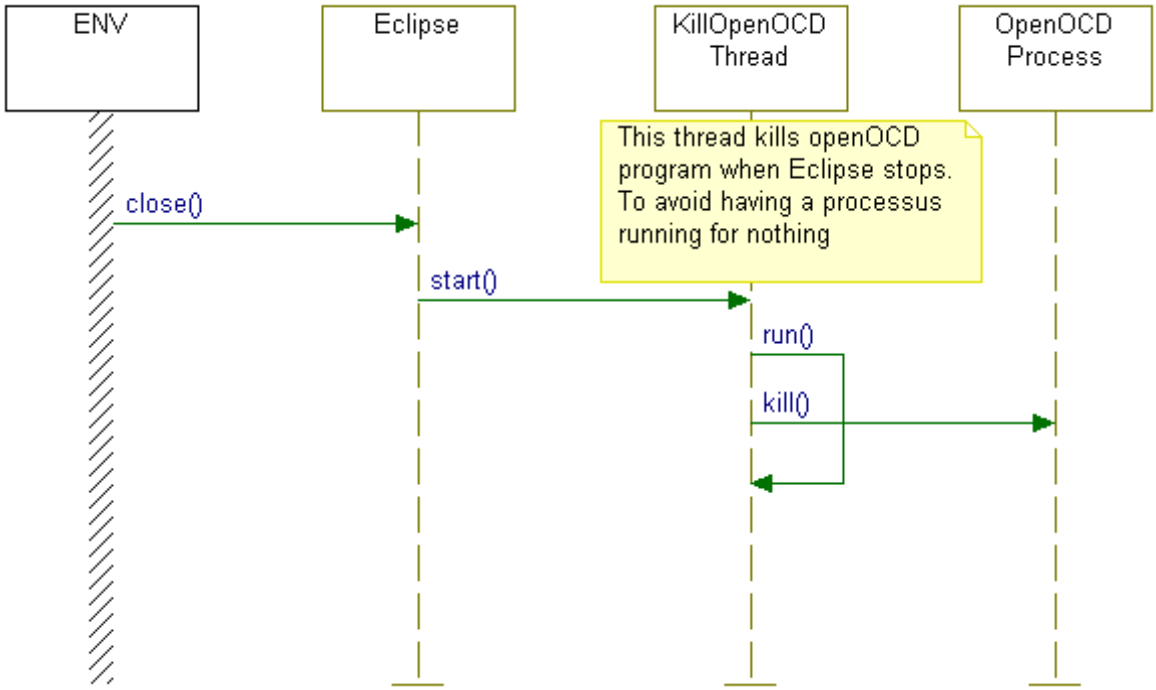
A l'intérieur de la méthode *run* un morceau de code, lui aussi synchronisé, lit la valeur de *stopThread* et la copie dans le flag *fin.* Ceci provoquera la fin du *thread ManagePrint.*

```
synchronized(this)
{
      fin = this.stopThread ;
}
```

Cette manière de faire peu paraître compliquée. C'est pourtant le seul moyen de garantir un arrêt correct du *thread.*

La classe KillOpenOCD

Comme expliqué plus haut cette classe permet de garantir la destruction du processus *openocd-ftd2xx.exe* et l'arrêt du *thread ManagePrint*. La méthode *run* de ce *thread* est automatiquement appelée par le *Runtime* lorsque l'IDE Eclipse est arrêté.



Annexe 1.5 Classes ManageOpenOCD et KillOpenOCD

La classe EmbeddedGDBCDIDebugger

Cette classe gère l'exécution du debugger. La méthode *createDebuggerSession* permet de faire appel aux méthodes *setOpenOCDInfo* et *restart* de la classe *ManageOpenOCD* juste avant de démarrer le debuggage sur la cible.

```
//get openOCD path and configuration file
openOCDConfigFile = launch.getLaunchConfiguration().getAttribute(
                  LaunchConfigurationConstants.ATTR_OPENOCD_CONFIG_FILE, "");

openOCDPath = launch.getLaunchConfiguration().getAttribute(
                       LaunchConfigurationConstants.ATTR_OPENOCD_PATH, "") ;

//give the openOCD path and config file to the Manager
ManageOpenOCD.getDefault().setOpenOCDInfo(openOCDPath, openOCDConfigFile)

//start openocd-ftd2xx.exe and ManagePrint thread
ManageOpenOCD .getDefault().restart() ;

//creat debugger session
dsession=super.createGDBSession(launch, exe.getPath().toFile(), monitor) ;
```

La notion de pointeur statique

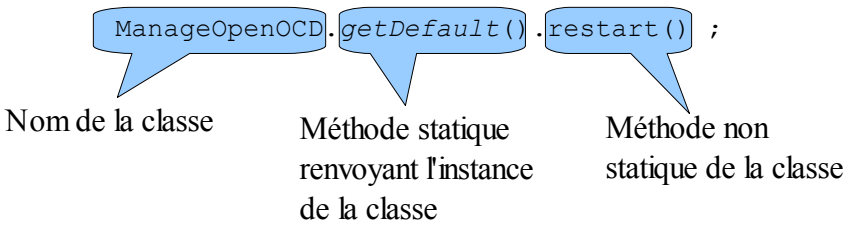
Les classes *ManageOpenOCD* et *OpenOCDConsole* peuvent être considérées comme des outils mis à disposition pour les autres classes du plugin *Embedded Hevs*. Pour utiliser les méthodes de ces deux classes la manière standard et de construire un objet de type *ManageOpenOCD* et d'appeler ensuite les méthodes désirées. Cette manière de faire devient problématique dès lors que deux classes veulent utiliser cet outil. Pour éviter de devoir déclarer un objet à chaque fois il est préférable que l'instance de la classe soit contenu à l'intérieur de celle-ci dans une variable statique accessible directement depuis l'extérieur.

Voici comment procéder : une instance statique est déclarée. Cette dernière est initialisée dans le constructeur. Pour accéder aux méthodes non statique de la classe il faut appeler la méthode *getDefault* de la classe.

Création de la classe :

```
private static ManageOpenOCD openOCD = null ;

public ManageOpenOCD ()
{
	if (openOCD == null)
	{
		openOCD = this ;
		System.out.println("Manager create") ;
	}
}

//return a pointer on the current class
public static ManageOpenOCD getDefault()
{
	return openOCD ;
}
```

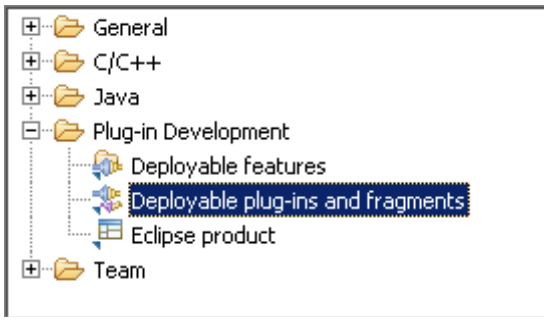
Utilisation d'une méthode de la classe *ManageOpenOCD*



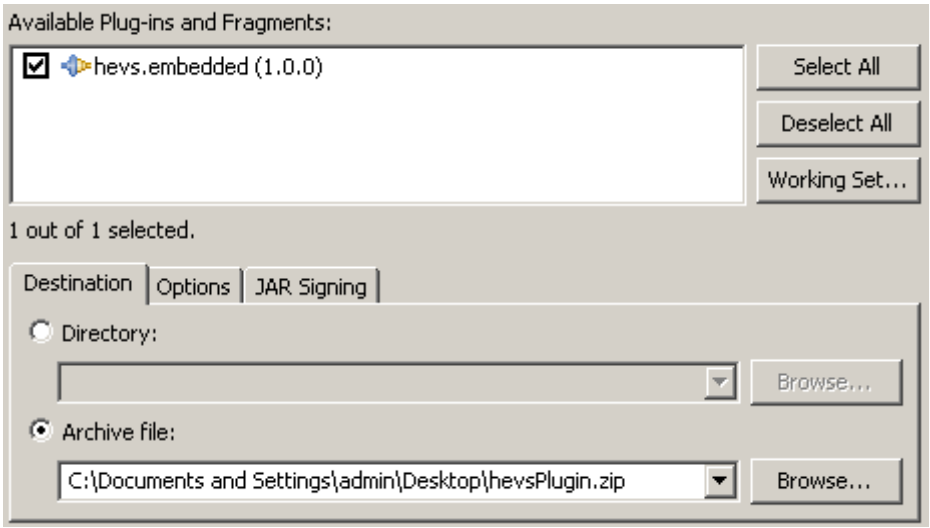
Deploiement du plugin

Lorsque toutes les extensions on été ajoutées et que les classes développées fonctionnent correctement il va falloir exporter le plugin sous forme de *.jar*. Lorsqu'un utilisateur désir utiliser le plugin *Embedded Hevs* il n'aura qu'à copier le fichier *.jar* dans le répertoire *plugins* d'Eclipse.

Pour exporter le plugin il faut sélectionner le menu *File – Export*. Apparaît ensuite la fenêtre *Export* permettant de choisir le format désiré. Pour un plugin standard il faut sélectionner *Deployable plug-ins and fragments*.



Après avoir cliqué sur le bouton *Next* une deuxième fenêtre apparaît il faut alors sélectionner le plugin à exporter et préciser l'emplacement ou l'archive *.zip* contenant le fichier *hevs.embedded_1.0.0.jar* sera enregistrée.



A noter que l'archive contient un dossier nommé *plugins* contenant le plugin lui-même. Il faudra donc extraire l'archive dans le dossier principal d'Eclipse pour ajouter le plugin développé.

Test grandeur nature

Installation vierge

Pour vérifier notre plugin *Embedded Hevs* nous allons effectuer un test général permettant de vérifier bien entendu notre plugin mais aussi les versions des autres outils utilisés.

Les outils/plugins utilisés sont :

- eclipse-SDK-3.3-win32.zip : l'IDE de base
- cdt-master-4.0.0.zip : plugin CDT 4.0
- embeddedcdt4.0-20070830.zip : plugin fournit par *Zylin* modifiant quelques aspects de CDT pour le debug embarqué
- hevsPlugin.zip : le plugin *Embedded Hevs*

Pour commencer, il faut extraire l'IDE de base sur le disque puis ajouter les plugins les uns après les autres. Une chose importante à faire est de démarrer Eclipse en ligne de commande en ajoutant le flag *-clean* pour recharger correctement les plugins ajoutés. Il faut effectuer cette opération qu'une seule fois, après l'ajout des plugins.

```
D:\eclipse>eclipse.exe -clean
```

Lorsque l'IDE est démarré nous allons vérifier que les plugins ajoutés figurent bel et bien dans la configuration. Pour ce faire il faut cliquer sur le menu *Help – About Eclipse SDK* puis cliquer sur le bouton *Plug-In Details.* Apparaît alors la liste des plugins installés. On peut constater que le plugin CDT est bien installé et que notre plugin figure aussi dans la liste.

	Eclipse.org	Http Service Registry Ext...	1.0.0.v2007...	org.eclipse.equinox.http.reg.
	Hevs	Hevs for Embedded Systems	1.0.0	hevs.embedded
	Eclipse.org	Help System Webapp	3.3.0.v2007...	org.eclipse.help.webapp

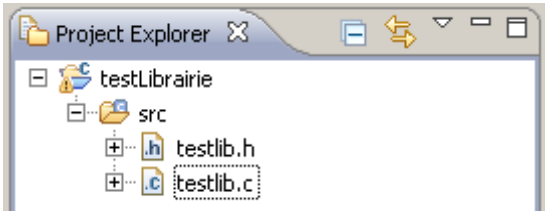
Test complet du plugin

Pour effectuer le test complet du plugin nous allons dans un premier temps créer un projet permettant de développer une librairie. Cette librairie contiendra simplement deux fichiers *.c* et *.h* mettant à disposition une fonction toute simple qui met une variable à zéro. Dans un deuxième temps un projet de type exécutable sera développé. Ce projet utilisera la fonction de la librairie implémentée plus haut.

Pour finir nous allons télécharger et exécuter le code en mode debug sur la carte ARMEBS3. On aura alors tester tous les aspects du plugin *Embedded Hevs*.

Création de la librairie

Après avoir créé un projet de type *Static Library* via le template *Embedded ARMEBS3*, nous allons lui ajouter deux fichiers un *header* et un fichier source :

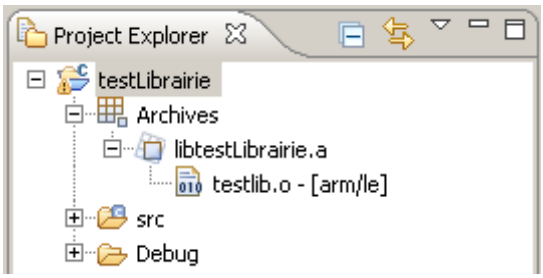


Nous allons ajouter au *.h* le prototype de la fonction *setToZero* et dans le *.c* le code de cette fonction.

```
//fichier testlib.h

#ifndef TESTLIB_H_
#define TESTLIB_H_

void setToZero (int *v) ;

#endif /*TESTLIB_H_*/
```

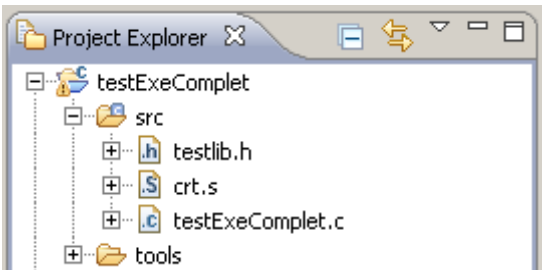
```
//fichier testlib.c

#include "testlib.h"

void setToZero (int *v)
{
	*v = 0 ;
}
```

Après avoir construit le projet, la librairie *libtestLibrairie.a* est générée. Elle contient l'objet *testlib.o* contenant le code de la fonction déclarée plus haut.



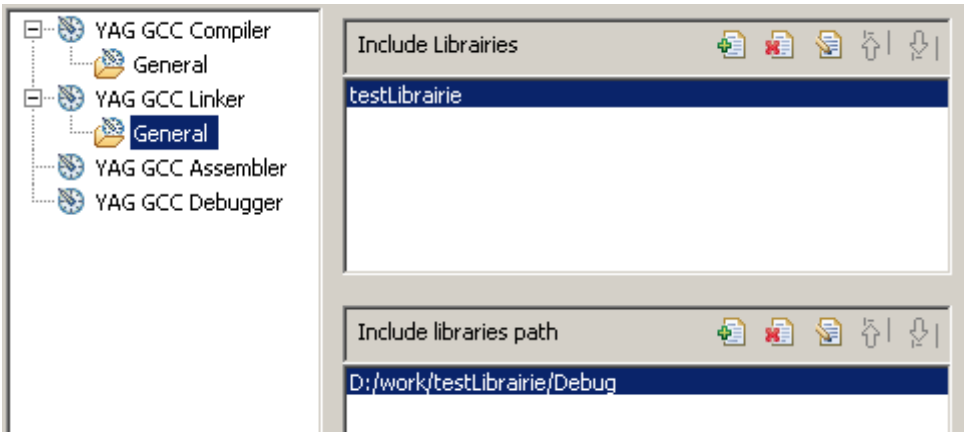
Création de l'exécutable

Pour utiliser la librairie *libtestLibrairie.a* nous allons créer tout d'abord un projet de type *Executable* via le template *Embedded ARMEBS3*.



Il faut bien entendu ajouter le fichier *testlib.h* aux fichiers sources du projet *testExeComplet*.

Il faut maintenant éditer les paramètres du projet *C* en ajoutant le path de la librairie développée ainsi que son nom pour que le linker puisse utiliser la fonction *setToZero*. Il faut impérativement utiliser des '/' et non pas des '\' dans le chemin de la librairie. En ce qui concerne le nom de cette même librairie il ne faut pas noter le préfixe *lib* et l'extension *.a*.



Pour utiliser cette fonction il faut maintenant éditer le *main* du programme. Il suffit d'inclure le fichier *testlib.h* et d'utiliser la fonction mise à disposition par la librairie. Voyons le programme très simple qui nous permettra de tester par la suite le debug.

```
#include "testlib.h"

int main ()
{
	int i = 0 ;

	while (1)
	{
		i++ ;

		if (i > 4)
			setToZero (&i) ;
	}

	return 0 ;
}
```

Pour finir, il suffit de construire l'exécutable *testExeComplet.elf*. Nous voyons dans la console que le *makefile* ajoute le path de la librairie et utilise cette dernière lors du link du programme :

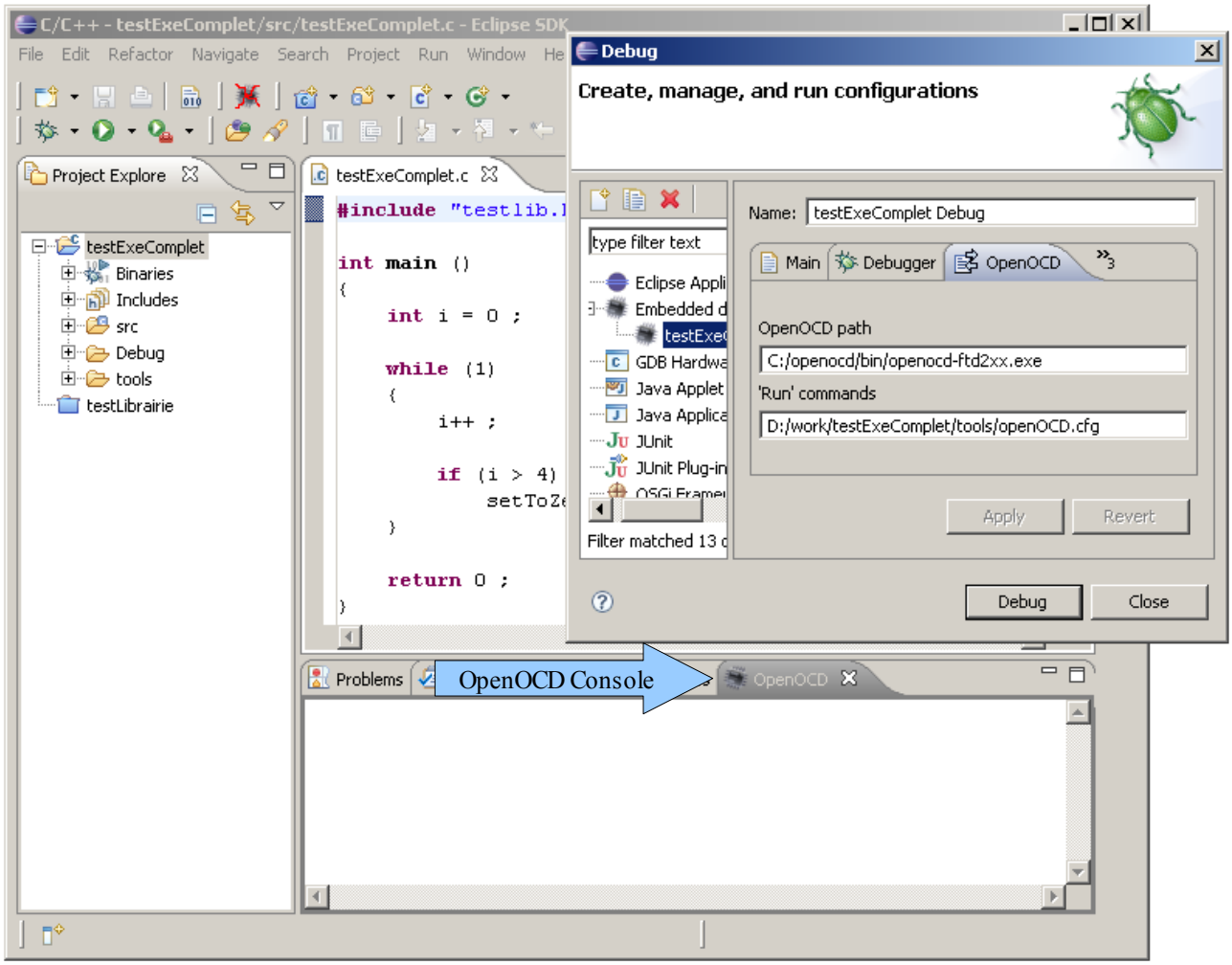
```
Building target: testExeComplet.elf
Invoking: YAG GCC Linker
arm-elf-gcc -LD:/work/testLibrairie/Debug
"D:\work\testExeComplet/tools/armebs3_ram.ld" -nostartfiles -Xlinker -M -Xlinker
-Map=map.map -o"testExeComplet.elf"  ./src/crt.o ./src/testExeComplet.o
-ltestLibrairie
Finished building target: testExeComplet.elf
```

ajout du path

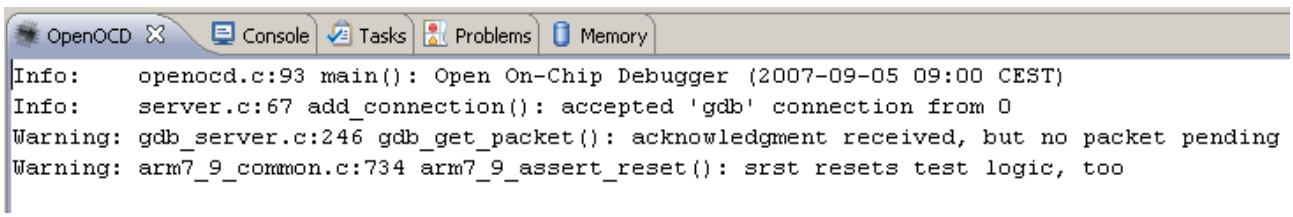
librairie utilisée

Debug sur la carte

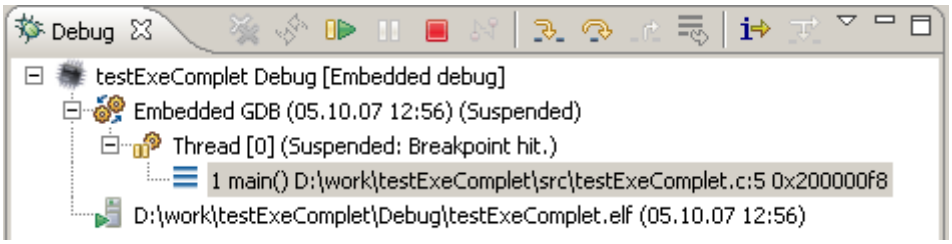
Après avoir effectué les mêmes opérations décrites sous le titre *Test de la configuration pour lancer le debug*, Nous allons pouvoir debugger la carte ARMEBS3. Il faut bien entendu que cette dernière soit alimentée et branchée au *PC* via la *JTAG Key*. Pour pouvoir visualiser les informations d'*OpenOCD* il faut encore ouvrir la console adéquate.



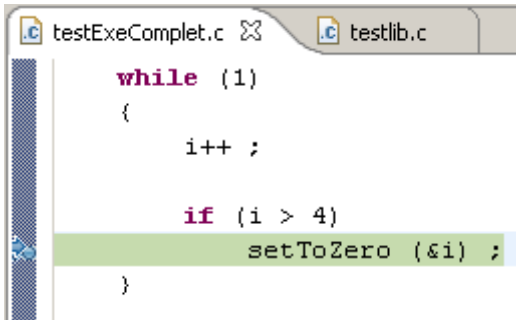
Lorsque la configuration du debug est correcte il suffit de cliquer sur le bouton *Debug*. La console *OpenOCD* affiche les informations récupérées depuis *openocd-ftd2xx.exe* et la vue spécifique au debug s'affiche. C'est à ce moment là que les commandes se trouvant dans le fichier *tools/gdbCmds* sont exécutées.



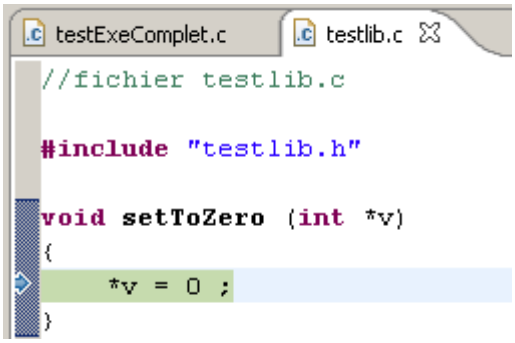
Plusieurs *widgets* sont affichées sur la vue relative au debug. Notamment la widget intitulée *Debug* qui permet de continuer, d'arrêter, et de faire des *step* après l'arrêt à un *breakpoint.*



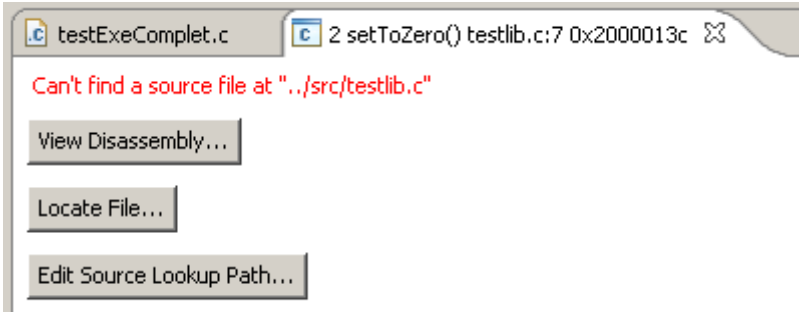
La widget central de cette vue affiche le code exécuté et permet de mettre des *breakpoints* aux endroits désirés.



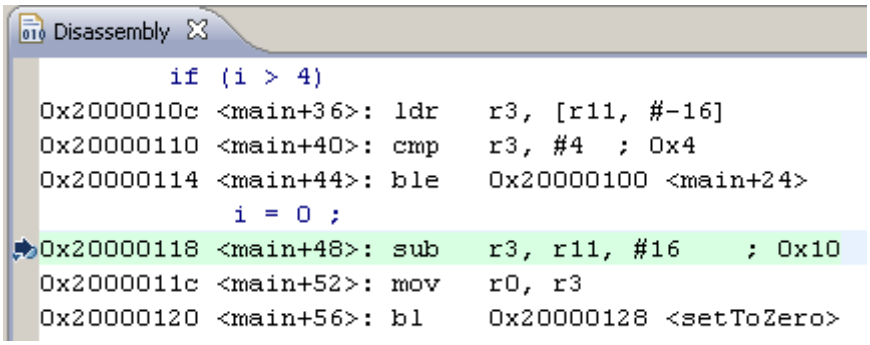
Voyons comment le debug va gérer l'appel de la fonction *setToZero*. En effet, le code de cette fonction est contenue dans la librairie. Si le projet *testLibrairie* est toujours ouvert dans le workspace, Eclipse est capable de récupérer le code depuis le projet pour l'afficher lorsqu'on l'exécute.



Bien entendu, si on ne possède pas les sources de la librairie (elle a été développée sur un autre poste par exemple) Eclipse informe que les sources sont introuvables et propose trois façon de palier au problème.

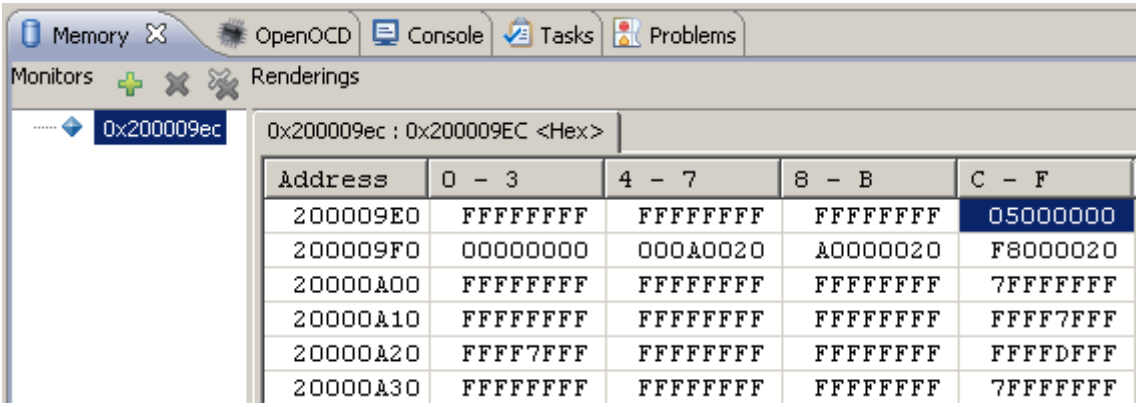


Nous pouvons encore mentionner deux autres *widgets* assez intéressantes. Il s'agit d'une part de la *widget Disassembly* permettant de voir le code en assembleur du programme (la traduction du C en assembleur) et d'autre part la *widget Memory* permet d'accéder en lecture et en écriture à la mémoire de l'ARMEBS3.

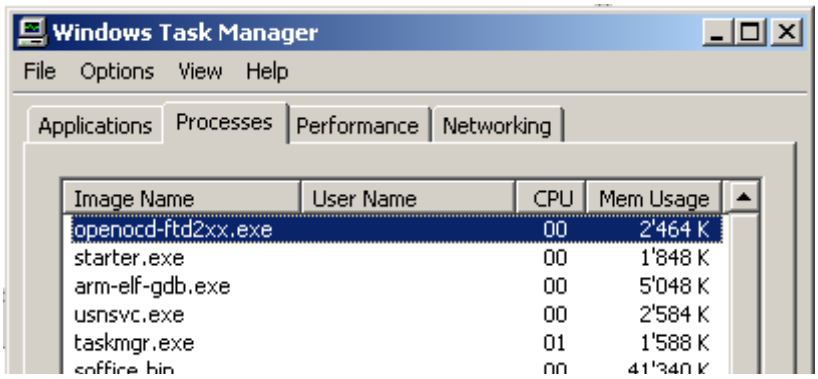


Ce widget donne un aperçu du code assembleur. On peu voir la traduction en assembleur des lignes de code rédigées en C.

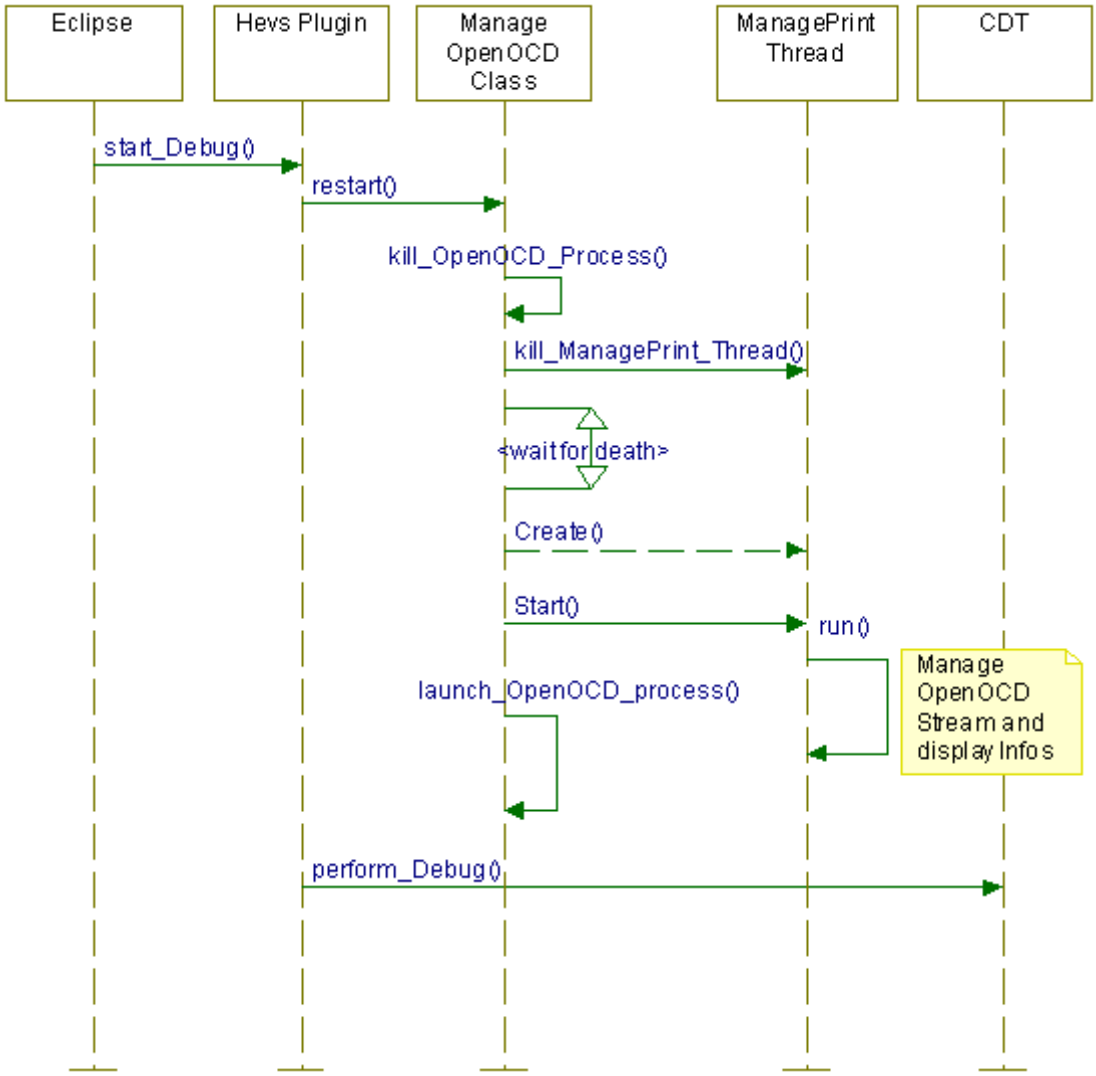
Le widget suivant permet d'accéder à la mémoire en ajoutant à gauche les emplacements que l'on désire observer/modifier.



Nous allons effectuer encore une petite vérification. Lorsque nous procédons au debug de la carte on peut constater que le programme *openocd-ftd2xx.exe* fait bien partie des Processus du système. Quand Eclipse est fermé ce processus est tué automatiquement par la *thread KillOpenOCD*.



Voici un résumé des opérations effectuées lors du debugging.



RTEMS sur l'ARMEBS3

La deuxième partie du travail qui m'a été proposé consiste à porter le système d'exploitation *RTEMS* sur la carte *ARMEBS3*. Comme expliqué plus haut cette carte a été développée par l'unité *infotronic* de la HES SO de Sion. Elle est utilisée notamment par les étudiants de troisième année pour les laboratoires de systèmes embarqués et de programmation en temps réel.

Aspects intéressants

- Multitasking, priority, timeslice
- Interruption : IRQ FIQ
- Allocation dynamique
- Timers, events, semaphores, queues, clock
- Networking : TCP/IP, UDP, TFTP, HTTPD
- Systèmes de fichiers : IMFS, FAT16, TFTP

A noté que l'environnement de développement est basé sur les outils GNU. Une toolchain spécifique est disponible pour compiler l'exécutif *RTEMS* puis le programme développé par l'utilisateur.

Installation de *VMware*

Pour démarrer une instance Linux sur une machine Windows il faut tout d'abord télécharger *VMware Server* et l'installer. Après son installation il suffit de télécharger l'image ISO de la distribution choisie. Pour ma part j'ai utilisé la distribution *Fedora Core*, cette dernière est sponsorisée par Redhat. Voici la marche à suivre pour créer une nouvelle machine virtuelle :

- Type de systèmes d'exploitation de la machine virtuelle : *Redhat Linux*.
- Disque virtuel : IDE 5.0GB (attention les disques virtuels SATA ne sont pas supportés)
- Mémoire RAM : 256Mo
- Réseau : commuté (direct)
- Son : pas de son
- USB : présent (permet de transférer des fichiers entre la machine virtuel (Linux) et le systèmes de base (Windows) via clé usb) pour monter la clé il faut cliquer sur le menu VM – Removable Device – USB Devices - <your device>

Maintenant que la machine virtuel est prête nous allons pouvoir installer la distribution choisie sur le disque virtuel. Pour ce faire il faut charger l'image ISO dans le CD-Rom virtuel :

- Edit virtual machine setting – CD-Rom (IDE 1:0) – Use ISO image

Il faut sélectionner le chemin vers l'image ISO téléchargée précédemment. Lorsque tout est bien configuré nous allons pouvoir démarrer la machine virtuelle en cliquant sur *Start this virtual machine.*

Lorsque l'on démarre pour la première fois sur l'image ISO il va falloir installer Linux sur le disque virtuel. Il suffit de se laisser guider par l'installateur et de choisir les options adéquates.

Installation des outils

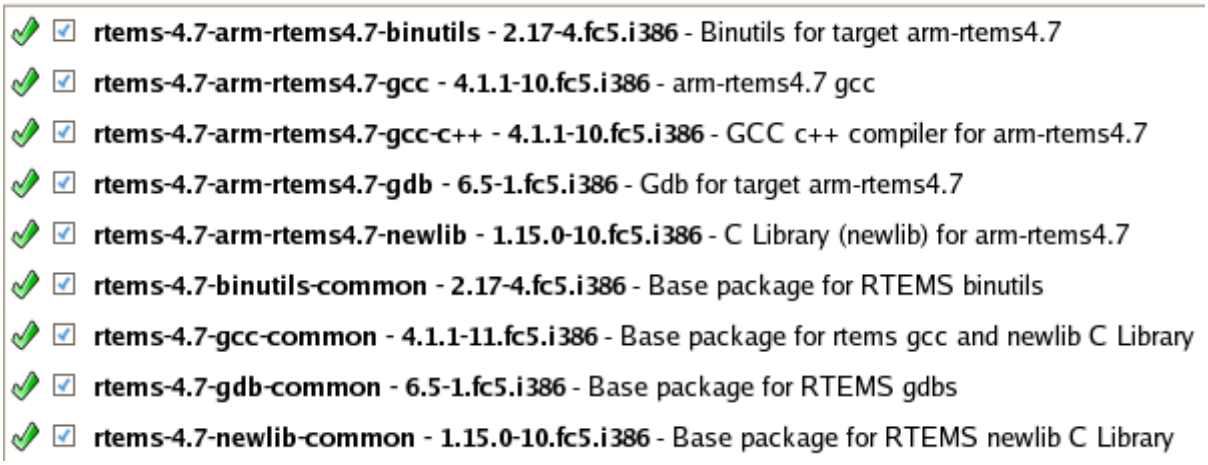
Pour Linux

La distribution que nous venons d'installer nécessite non seulement les outils permettant de compiler le kernel RTEMS mais aussi les outils de bases suivants :

- autoconf 2.6, automake 1.1
- binutils 2.17
- gcc 4.1.1 for C/C++ with newlib 1.15.0
- gdb 6.5

Pour le kernel RTEMS

Nous allons maintenant télécharger les outils nécessaires à la compilation du noyau de *RTEMS*. Ces outils sont représentés par des paquet *rpm* que nous pourrons installer par la suite. Ces derniers se trouvent à l'adresse suivante : http://www.rtems.com/ftp/pub/rtems/linux/. Il faut bien entendu sélectionner le répertoire correspondant à la distribution utilisée ainsi que la version de RTEMS désirée, la 4.7 est la dernière version stable. les paquets qui nous intéressent sont les suivants :



L'installation de ces paquets nécessite les droit root. Il suffit d'entrer la commande suivante pour installer les paquets les uns après les autres.

```
rpm -i rtems-4.7-gcc-common-4.1.1-11fc5.i.386.rpm
```

A noter que les paquets xxx-common-xxx doivent être installés avant les paquets spécifiques au cpu ARM. Toutefois si une dépendances n'est pas résolue l'installateur vous indiquera quel paquet installer en premier. Après l'installation de ces outils on retrouve dans le dossier */opt/rtems-4.7/bin/* les exécutables nécessaires à la compilation de noyau *RTEMS*. Comme pour la compilation des *standalones* à travers Eclipse ces outils sont dérivés de la collection GCC et portent le préfixe *arm-rtems4.7*. Pour que ces outils soient atteignables de n'importe quel dossier il faut ajouter au path le répertoire suivant : */opt/rtems-4.7/bin/*

```
export PATH=/opt/rtems-4.7/bin/:$PATH
```

Modifications des fichiers sources

Lorsque les outils nécessaires sont correctement installés il est maintenant possible de compiler l'exécutif *RTEMS*. Pour ce faire il faut bien entendu télécharger la dernière version stable qui est, à ce jour, la version 4.7.1 (http://www.rtems.com/ftp/pub/rtems/4.7.1/rtems-4.7.1.tar.bz2).

les fichiers sont regroupés et compressés dans une archive *.tar.bz2*. Pour décompresser cette archive il faut entrer la commande suivante

```
tar xvf rtems-4.7.1.tar.bz2
```

Pour que l'exécutif soit compatible avec l'ARMEBS3 il va falloir vérifier quelques points importants. Il faut tout d'abord savoir que *RTEMS* est compatible avec bon nombre de processeurs et de BSP. C'est pourquoi il faut bien vérifier que les modifications apportées soient en relation avec le processeur et le BSP utilisés. Voici les fichiers importants :

- `cpukit/libcsupport/src/write.c - read.c`

  Ces fichiers permettent de surcharger les fonctions *write* et *read* utilisées par *printf* et *scanf*. Nous utilisons le port RS232 de la carte pour visualiser et envoyer les informations via un terminal connecté au port COM. Le code permettant de rediriger ces informations a été écrit par Monsieur Sartoretti, je l'ai simplement récupéré tel quel. A noter que les fichiers suivant doivent aussi être installés au même endroit : *`hevstypes.h`*, *`AT91RM9200.h`*, *`arm_io.h`*

  Annexe 2.1 arm_write.c, arm_read.c

- `c/src/lib/libbsp/arm/csb337/start/start.S`

  Ce fichier est comparable au *crt.s* utilisé dans la première partie du projet. Il regroupe les informations concernant le mode d'exécution et la table des vecteurs. De plus, ce fichier gère l'appel de la fonction *bsp_start* permettant de démarrer l'exécution du noyau.

- `c/src/lib/libbsp/arm/csb337/startup/bspstart.c`

  Les fonctions regroupées dans ce fichier démarrent le management de la mémoire (*heap)* et des interruptions

- `c/src/lib/libbsp/arm/csb337/startup/linkcmds`

  Ce fichier est comparable au fichier *.ld* utilisé pour programmer l'ARMEBS3 en *standalone*. Il spécifie l'origine de la RAM ainsi que sa taille, les sections utilisées et leur emplacement dans la mémoire. Les modifications apportées concernent la taille de la mémoire ainsi que son origine

  Annexe 2.2 linkcmds

- `c/src/lib/libbsp/arm/csb337/include/bsp.h`

  Le BSP utilisé est à la base destiné à une autre carte basée sur le même processeur que l'ARMEBS3. Il faut modifier ce fichier qui déclare une constante appelée BSP_MAIN_FREQ correspondant au quartz de 20 Mhz que nous avons sur notre carte.

  Annexe 2.3 bsp.h

Plusieurs autres fichiers feront leur apparition lors du test des outils mis à disposition par l'exécutif *RTEMS*.

Compiler l'exécutif RTEMS

Après avoir modifié les fichiers qui nous intéressent, nous allons maintenant procéder à la configuration du *makefile*. En d'autre terme nous allons générer un fichier *makefile* spécifique en passant au script *configure* toutes les informations nécessaires à l'os. Pour avoir la liste complète des options de *configure* il suffit de taper *configure --help*. Pour ce faire nous allons créer deux nouveaux dossiers au même niveau que le dossier contenant les sources de l'os. Le premier intitulé simplement *scriptForArm* contiendra les *makefiles* nécessaires à la compilation de l'os. Le deuxième appelé *rtemsForArm* contiendra l'exécutif compilé spécifique au processeur at91rm9200 et au BSP csb337.

BSP csb337

Il faut savoir tout d'abord qu'un *Board Support Package* est un logiciel de bas niveau de support de cartes-mères, il s'intègre entre l'OS et le hardware de la carte mère. Il n'existe pour l'instant pas de BSP spécifique à notre ARMEBS3. Nous utilisons donc le BSP *csb337* qui est, à la base, prévue pour une carte munie du même processeur que la notre. Cette carte est fournie par la société *Cogent Computer Systems*.

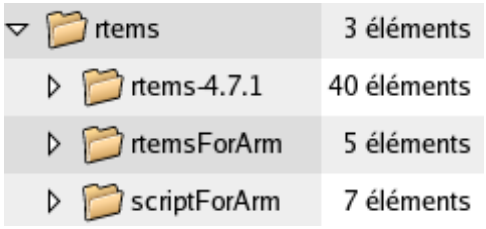
Modification du fichier *make/custom/csb337.cfg*

Le fichier de configuration en rapport avec le BSP utilisé déclare une variable très intéressante se rapportant aux falgs utilisés lors de la compilation du kernel. Pour compiler le kernel nous allons modifier ces flags pour que les informations relatives au debuggage ne soit pas générées. Il est préférable d'utiliser cette méthode pour obtenir un kernel plus léger. Il faut enlever le falg *-g* Voici la variable à modifier :

```
CFLAGS_OPTIMIZE_V = -O2
```

Configuration

Pour générer les *makefiles* désirés nous allons nous placer dans le dossier *scriptForArm* et entrer la commande suivante :




```
../rtems-4.7.1/configure --target=arm-rtems4.7 --disable-posix
                         --disable-networking --disable-cxx
                         --enable-rtemsbsp=csb337
                         --prefix=/home/joel/rtems/rtemsForArm
```

Les informations les plus importantes concernant cette commande sont les suivantes :

- target : on spécifie quelle toolchain utiliser pour compiler les fichiers sources. Le préfix *arm-rtems4.7* est retrouvé dans le nom des outils utilisés :*arm-rtems4.7-gcc, arm-rtems4.7-as* etc

- enable-rtemsbsp : on indique ici quel BSP utiliser pour compiler l'exécutif

- prefix : représente le répertoire d'installation de l'exécutif compilé. Lorsque l'on entrera la commande *make install* les headers et les librairies compilées viendront se copier dans ce répertoire.

Après avoir exécuté cette commande le dossier *scriptForArm* contient toutes les informations nécessaires à la compilation de l'exécutif en rapport à la toolchain et au BSP désiré. Le *makefile* est maintenant correctement configuré pour la compilation.

Si tout s'est bien déroulé le script doit se terminer par les lignes suivantes :

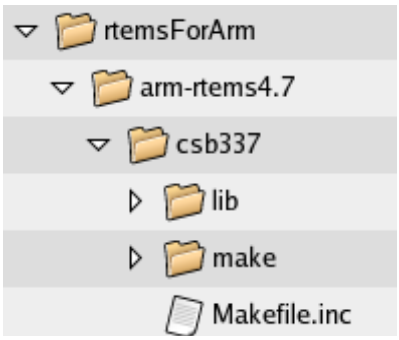
```
config.status: creating Makefile
config.status: creating make/Makefile
[joel@localhost scriptForArm]$
```

A noter qu'il ne sert à rien de compiler l'exécutif si la configuration effectuée ci-dessus à échoué. Il faut vérifier que le dossier */opt/rtems-4.7.1/bin* est bien inclut dans le path. Pour vérifier que c'est bien le cas la commande *arm-rtems4.7-gcc* doit être reconnue par le système.

Compilation

Pour lancer la compilation il suffit d'entrer la commande *make* dans ce même répertoire *scriptForArm*.

La compilation de l'exécutif dure quelques minutes. Si tout ce passe bien, on retrouve dans le répertoire *scriptForArm/arm-rtems4.7/csb337/lib* les librairies compilées et dans le dossier *include* de ce même répertoire les *headers* mis à disposition *par RTEMS*. Pour récupérer l'exécutif compilé sans les fichiers de configuration il suffit d'entrer la commande *make install*. Les libraires et les *headers* seront alors copiés dans le répertoire *rtemsForArm*.



Le répertoire *rtemsForArm* visible ci-contre regroupe au prermier coup d'oeil un certain nombre d'information : on retrouve la toolchain utilisée, le BSP utilisé ainis que le fichier *Makefile.inc* qui permet de lier l'exécutif *RTEMS* (le noyau) avec le programme développé par l'utilisateur.

Le passage sous Windows

Maintenant que l'exécutif est à notre disposition nous allons repasser du côté Windows pour tester ce dernier sur l'ARMEBS3. Pour ce faire nous allons copier le dossier *rtemForArm* sous D:/ et ajouter la variable d'environnement RTEMS_MAKEFILE_PATH pointant vers le répertoire contenant le makefile de l'exécutif : *Makefile.inc.* Cette manière de faire permet à tous les programmes développés dans le futur de savoir où se trouve le kernel *RTEMS*

Il faut aussi copier le dossier *rtems-4.7.1* contenant les fichiers sources de l'OS au même endroit que le dossier *rtemsForArm.* Ce dernier est nécessaire si l'on désire accéder aux fonctions de l'OS lors du debuggage.

Il faut maintenant modifier le fichier *Makefile.inc.* Sous Linux, le répertoire de l'exécutif se trouve sous */home/joel/rtems/rtemsForArm/.* Le répertoire contenant les librairies et les headers se trouve dans le répertoire *arm-rtems4.7 :*

```
RTEMS_BSP = csb337

prefix= /home/joel/rtems/rtemsForArm/
exec_prefix= /home/joel/rtems/rtemsForArm/arm-rtems4.7
```

Il faut remplacer ces deux lignes par les chemins vers ces mêmes répertoires sous Windows :

```
RTEMS_BSP = csb337

prefix = D:/rtemsForArm/
exec_prefix = D:/rtemsForArm/arm-rtems4.7
```

Nous allons maintenant modifier la variable *CFLAGS_OPTIMIZE_V* contenue dans le fichier *make/custom/csb337.cfg* permettant de gérer les flags d'optimisation et de debug. Ce fichier contient le flag *-O2* que nous avons ajouté avant de compiler le kernel. Nous allons maintenant remplacé ce flag par *-O0 -g* pour pourvoir debugger correctement le code rédigé par l'utilisateur final.

```
CFLAGS_OPTIMIZE_V = -O0 -g
```

Dans ce même *makefile* on voit apparaître les outils de la toolchain *arm-rtems4.7* qui vont être utilisés pour compiler l'exécutable final, le noyau et le code de l'utilisateur. Il faut donc installer ces outils sur la machine Windows (http://www.rtems.com/ftp/pub/rtems/windows/4.7/rtems4.7-arm-3.exe). Après avoir installé ces outils il faut bien entendu ajouter au path le dossier contenant la toolchain pour que le *makefile* sache où les trouver lors de la compilation du programme.

Premier test de RTEMS

Des exemples relativement simples sont mis à disposition pour tester notre OS. Tout d'abord, il faut les télécharger à l'adresse suivante : http://www.rtems.com/ftp/pub/rtems/4.7.1/examples-4.7.1.tar.bz2 et les décompresser sur le disque. Pour effectuer un premier test avec *RTEMS* nous allons compiler un exemple fournit, le télécharger sur la carte et le debugger. Pour savoir comment faire, il faut se reporter au chapitre *Debugger un .elf avec ces outils*. Nous allons tester l'exemple s'intitulant *yield_flash*. Ce programme est composé de deux tâches ayant la même priorité et se partageant le temps processeur. Pour compiler l'exécutable et l'OS il suffit de se rendre dans le dossier *yield_flash* et d'entrer la commande *make*. A partir de ce moment, le programme en lui-même sera compilé et linké avec le kernel *RTEMS*. En effet, le *makefile* de l'exemple inclut le *Makefile.inc* relatif au kernel, voici un aperçu de ces deux fichiers.

Makefile de l'application

```
EXEC=test.exe
PGM=${ARCH}/$(EXEC)

MANAGERS=io

CSRCS = test.c
COBJS_ = $(CSRCS:.c=.o)
COBJS = $(COBJS_:%=${ARCH}/%)

include
$(RTEMS_MAKEFILE_PATH)/Makefile.inc
include $(RTEMS_CUSTOM)
include $(PROJECT_ROOT)/make/leaf.cfg

LIBS = -lrtemsall -lc
#...
```

Voici une partie du *makefile* de l'application *yiels_flash* il faut noter qu'on spécifie ici le nom de l'exécutable généré ainsi que les fichiers sources de l'application. Une partie importante est de configurer correctement la variable *MANAGERS* qui permet de spécifier si l'on utilise des semaphores, des messages (queues) des partitions ou des timers.
On inclut encore dans ce *makefile* d'autres informations concernant le BSP utilisé ainsi que le *makefile* relatif au kernel de *RTEMS*. Il faut pour finir spécifier les libraires utilisées : *rtemsall* et *c*

Makefile du kernel *rtemsForArm\arm-rtems4.7\csb337*

```
RTEMS_BSP = csb337

prefix = D:/rtemsForArm/
exec_prefix = D:/rtemsForArm/arm-
rtems4.7

CC_FOR_TARGET = arm-rtems4.7-gcc --pipe
AS_FOR_TARGET = arm-rtems4.7-as
AR_FOR_TARGET = arm-rtems4.7-ar
LD_FOR_TARGET = arm-rtems4.7-ld

RTEMS_CUSTOM =
$(prefix)/make/custom/$(RTEMS_BSP).cfg
PROJECT_ROOT = $(prefix)
RTEMS_HAS_CPLUSPLUS = no
RTEMS_HAS_POSIX_API = no
```

Comme expliqué ci dessus, le *makefile* concernant le kernel regroupe le chemin vers l'OS compilé sur linux, il définit encore le BSP utilisé. Plus bas sont regroupées les informations relatives à la toolchain utilisée (*arm-rtems4.7*). On aperçoit encore que plusieurs variables sont déclarées, ces dernières servent à « customiser » la compilation du kernel et du programme utilisateur en définissant notamment des flags spécifiques à notre processeur. On définit encore quelques informations relatives aux outils utilisés, C++, POSIX...

Annexe 2.4 Makefiles

Compilation et téléchargement

Comme expliqué plus haut, la compilation s'effectue simplement en entrant la commande *make* dans le répertoire de l'exemple. Si tout se passe bien, les informations suivantes sont affichées

```
C:\WINDOWS\system32\cmd.exe
C:\examples-4.7.1\yield_flash>make
test -d o-optimize || mkdir o-optimize
arm-rtems4.7-gcc --pipe -BD:/rtemsForArm/arm-rtems4.7/csb337/lib/ -specs bsp_spe
cs -qrtems    -g -Wall  -O2 -g  -g     -mcpu=arm920 -mstructure-size-boundary=8
   -c    -o o-optimize/test.o test.c
arm-rtems4.7-gcc --pipe -BD:/rtemsForArm/arm-rtems4.7/csb337/lib/ -specs bsp_spe
cs -qrtems    -g -Wall  -O2 -g  -g     -mcpu=arm920 -mstructure-size-boundary=8
      -o o-optimize/test.exe  o-optimize/test.o    D:/rtemsForArm/arm-rtems4.7/cs
b337/lib/no-dpmem.rel D:/rtemsForArm/arm-rtems4.7/csb337/lib/no-event.rel D:/rte
msForArm/arm-rtems4.7/csb337/lib/no-msg.rel D:/rtemsForArm/arm-rtems4.7/csb337/l
ib/no-mp.rel D:/rtemsForArm/arm-rtems4.7/csb337/lib/no-part.rel D:/rtemsForArm/a
rm-rtems4.7/csb337/lib/no-signal.rel D:/rtemsForArm/arm-rtems4.7/csb337/lib/no-t
imer.rel D:/rtemsForArm/arm-rtems4.7/csb337/lib/no-rtmon.rel
arm-rtems4.7-nm -g -n o-optimize/test.exe > o-optimize/test.num
arm-rtems4.7-size o-optimize/test.exe
   text    data     bss     dec     hex filename
  93152    3548   48704  145404   237fc o-optimize/test.exe

C:\examples-4.7.1\yield_flash>
```

On aperçoit ici que l'exécutable contenant le programme et l'OS est intitulé *test.exe*. A la fin de la construction de l'exécutable le script nous affiche encore les différentes section du programme avec leur taille respective. A partir de maintenant, nous allons effectuer les opérations décrites dans le chapitre *Debugger un .elf avec ces outils* pour télécharger le code exécutable sur la carte ARMEBS3. Après ça, nous allons mettre un *breakpoint* sur le point d'entrée des deux tâches déclarées par le programme, Puisqu'elles ont la même priorité elles doivent s'exécuter l'une après l'autre : voici un aperçu du test effectué :

```
Start address 0x20100020, load size 96700
Transfer rate: 193400 bits/sec, 363 bytes/write.
(gdb) b task1
Breakpoint 1 at 0x2010032c: file test.c, line 27.
(gdb) b task2
Breakpoint 2 at 0x2010031c: file test.c, line 37.
(gdb) c
Continuing.

Breakpoint 1, task1 () at test.c:27
27          printf("task1 is generating lots of output\n");
(gdb) n
28          rtems_task_wake_after(RTEMS_YIELD_PROCESSOR);
(gdb) c
Continuing.

Breakpoint 2, task2 () at test.c:37
37          rtems_task_wake_after(RTEMS_YIELD_PROCESSOR);
(gdb) c
Continuing.

Breakpoint 1, task1 () at test.c:27
27          printf("task1 is generating lots of output\n");
(gdb) _
```

break task1
break task2
stop in task1
stop in task2
stop in task1

Comme prévu, les tâches se partagent le processeur en round-robin puisqu'elles possèdent la même priorité. Si la task1 avait une priorité supérieure à la task2 elle monopoliserait le cpu.

Test des directives RTEMS

Nous savons maintenant que le code s'exécute correctement sur l'ARMEBS3. Pour expérimenter les différents outils mis à disposition par *RTEMS* un certain nombre de tests et d'exemples ont été rédigés et mis en annexe. Le tableau ci-dessous résume les tests effectués.

Outils	Directives	Remarques

tasks	`rtems_task_create`
	`rtems_task_start`
	`rtems_task_wake_after`
	`rtems_task_wake_when`
		Les fonctions listées ci-contre sont les directives de base fournies par *RTEMS*. Ces dernières fonctionnent correctement, à noter qu'un grand nombre d'autres fonctions relatives au tâches sont disponibles. *Annexe 2.**

I/O	`printf, scanf, fflush`	La surcharge des fonctions *read* et *write* ajoutées avant la compilation du kernel redirigent correctement les flux vers la sortir/entrée RS232. *Annexe 2.0, 2.1*

semaphores	`rtems_semaphore_create`
	`rtems_semaphore_obtain`
	`rtems_semaphore_release`
		Les fonctions relatives aux sémaphores fonctionnent correctement. L'exemple mis en annexe permet de partager l'output standard entre deux tâches. *Annexe 2.5*

queues	`rtems_message_queue_create`
	`rtems_message_queue_send`
	`rtems_message_queue_receive`
		Les message queues misent à disposition font bien leur travail. Leur utilisation s'apparente à celle de *Threadx* (wait, timeout...). *Annexe 2.6*

timers	`rtems_timer_create`
	`rtems_timer_fire_after`
	`rtems_timer_reset`
		Les timers de *RTEMS* sont basés sur l'horloge interne du cpu, cette manière de faire garanti la survie du timer y compris lorsque le cpu se met en veille. Cependant les timers software seront légèrement moins précis que les timers hardware. *Annexe 2.7. + printScreen*

events	`rtems_event_send` `rtems_event_receive`	Les événements sont regroupés en *set* de 32 événements. Les deux directives ci-contre permettent d'envoyer et d'attendre sur des événements. Cet outil est correctement implémenté sur notre cpu. *Annexe 2.8 et 2.9*
interruptions	`BSP_install_rtems_irq_handler`	La mise en place d'interruptions fonctionne correctement, cependant son implémentation est légèrement plus fastidieuse par rapport aux autres outils. On utilise la structure `rtems_irq_connect_data` pour mettre

		en place l'interruption. Cette structure contient notamment le numéro de l'interruption et la fonction à appeler lorsque cette dernière arrive. Les informations concernant la priorité de l'interruption ainsi que la détection de cette dernière (flan montant, descendant) doivent être ajoutées à la main. *Annexe 2.9 + printScreen*

interruptions events	`BSP_install_rtems_irq_handler`
	`rtems_event_send`
	`rtems_event_receive`
		La meilleur façon d'effectuer une opération lorsqu'une interruption arrive est d'envoyer un événement et de le traiter par la suite. Cette façon de faire permet de séparer l'interruption hardware de l'exécution software. Nous verrons dans les annexes que cette manière de faire est un peu moins rapide mais toujours très efficace. *Annexe 2.10 + printScreen*

clock manager	`rtems_clock_set` `rtems_clock_get`	*RTEMS* met à disposition une horloge système. Elle permet notamment d'endormir et de réveiller des tâches à certaine date. Bien entendu l'heure système est accessible en tout moment par n'importe quelle tâche. Le comportement du clock manager est correct sur l'ARMEBS3. *Annexe 2.11*

Partitions	`rtems_partition_create`
	`rtems_partition_delete`
	`rtems_partition_get_buffer`
	`rtems_partition_return_buffer`
		RTEMS permet d'allouer dynamiquement de la mémoire. Il suffit de déclarer une partition puis d'appeler les fonctions renvoyant des buffers de taille fixe. La gestion de ce type de mémoire fonctionne correctement. *Annexe 2.12*

Network	`rtems_bsdnet_initialize_network` `rtems_bsdnet_synchronize_ntp`	Le BSP utilisé met à disposition un certain nombre de fonctions permettant d'utiliser l'interface Ethernet de notre cpu. Cette dernière fonctionne correctement sur l'ARMEBS3. Cependant quelques changements ont été apportés pour que tout fonctionne correctement. La page ci-dessous donne quelques informations supplémentaire concernant le network. *Annexes 2.13 2.14*

Le réseau avec RTEMS

Pour utiliser l'interface réseau de *RTEMS,* il faut bien évidemment recompiler l'exécutif en ajoutant le falg *enable-network.* La commande permettant de configurer le kernel avant de le compiler est la suivante :

```
../rtems-4.7.1/configure --target=arm-rtems4.7 --disable-posix
                         --enable-networking --disable-cxx
                         --enable-rtemsbsp=csb337
                         --prefix=/home/joel/rtems/rtemsForArm
```

Le makefile

Les makefiles utilisés lors de développements réseaux sont légèrement différents de ceux utilisés pour une application locale. En effet les applications réseaux ont besoin de plus de mémoire et doivent utiliser des librairies supplémentaires. Les exemples officiels fournis permettent de mieux comprendre leur subtilité.

Le fichier *networkconfig.h*

Chaque exemples est fourni avec un fichier permettant de configurer *RTEMS* par rapport au réseau que nous utilisons. Bien entendu, il est possible de spécifier sont adresse IP ou MAC mais aussi l'adresse IP du serveur NTP ou encore la priorité des tâches concernant le réseau. Ce fichier doit impérativement être configuré avant de pouvoir utiliser les outils relatifs au réseau.

La MAC ADDRESS

Après plusieurs essais infructueux, nous nous sommes rendus compte avec le logiciel *Ethereal* que l'adresse MAC de la cible n'était pas transmise correctement dans les paquets IP ou UDP. Nous avons dû reprogrammer les registres contenant cette adresse avant d'appeler la fonction `rtems_bsdnet_initialize_network`. Le problème vient du fait que l'interface JTAG, lors d'une exécution en debug, peut faire un reset des informations configurées par le u-boot de la cible.

Poursuite du travail

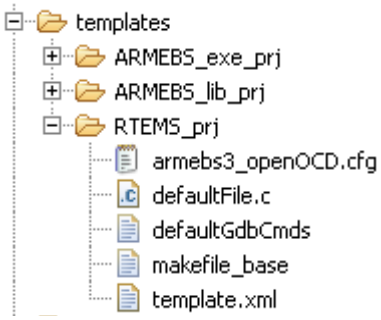
Pour que le portage de *RTEMS* soit complet, il serait intéressant de pouvoir envoyer des données Ethernet « à la main », c'est à dire en construisant les paquets avec les adresses source et destination, les données... Tout en réagissant sur les events envoyés lorsque de nouveaux paquets arrivent etc. De plus, il faudra porter les différentes mezzanines développées par l'unité *Infotronics* de l'école sur ce nouvel OS.

RTEMS sur ECLIPSE

Nous savons maintenant que cet OS fonctionne correctement sur l'ARMEBS3. Pour que son utilisation soit plus agréable aux utilisateurs nous allons ajouter quelques extensions au plugin *Embedded Hevs* pour pouvoir télécharger et debugger l'exécutable sur la cible. Comme expliqué ci-dessus la gestion des makefiles est assez compliquée. En effet, le makefile de l'exécutable développé est séparé de celui du noyau. De plus la toolchain utilisée est déclarée à l'intérieur du makefile du noyau. Sans compter les différents fichier relatifs au BSP utilisé. Nous avons décidé au final de laisser le soin à l'utilisateur de gérer le makefile de l'exécutable final. Les élèves qui utiliseront ce plugin auront donc l'opportunité de mieux connaître comment les makefiles sont gérés et de mettre les mains dans le « cambouis » lorsqu'un problème arrive. Bien entendu, le plugin charge un makefile et un fichier source par défaut ainsi que les fichiers *gdbCmds* et *openOCD.cfg* permettant de démarrer *openOCD* et d'envoyer les commandes à *gdb*.

Ajout du template

Après avoir ouvert le projet de développement de plugin nous allons ajouter le template permettant d'automatiser le chargement des fichier de base d'un projet de type *RTEMS on ARMEBS3*.



Nous voyons ci-contre le nouveau template ajouté. Celui ci est composé de cinq fichiers : un fichier source et un makefile standards, un fichier de configuration pour *openOCD* ainsi qu'un fichier contenant les commandes à envoyer à *gdb*.

Le dossier *RTEMS_prj* contient aussi le fichier *template.xml*. Ce dernier contient les actions à réaliser lors de la création d'un projet *RTEMS on ARMEBS3*.

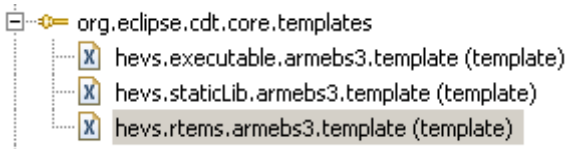
Ci dessous un aperçu du fichier *template.xml*. L'ajout des fichiers de base d'un projet *RTEMS* s'effectue de la même manière que dans la première partie.

```
<process type="org.eclipse.cdt.core.AddFiles">
<simple name="projectName" value="$(projectName)"/>
<complex-array name="files">
        <element>
                <simple name="source" value="defaultFile.c"/>
                <simple name="target" value="osEntry.c"/>
                <simple name="replaceable" value="true"/>
        </element>
...
```

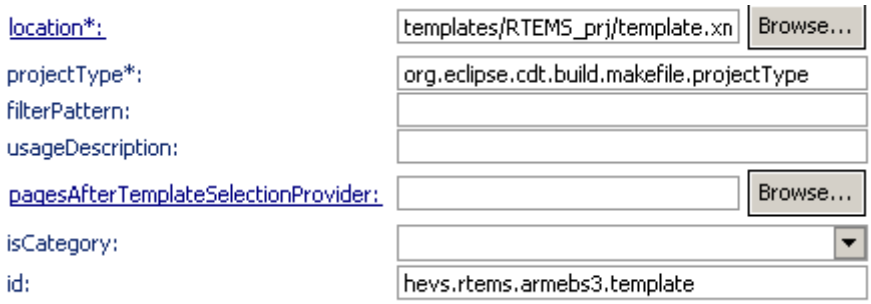
Ajout des extensions

org.eclipse.cdt.core.templates

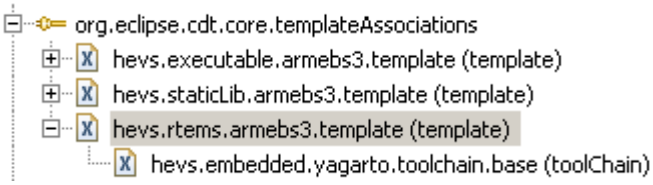
La première extension à ajouter permet de déclarer un nouveau template en relation avec les fichiers ajoutés plus haut. En guise de *projectType* nous lui donnons la valeur *org.eclipse.cdt.build.makefile.projectType* cela permet de gérer notre makefile nous-même. Pour rappel le *projectType* utilisé dans la première partie pour développer un exécutable dont le makefile est géré automatiquement était *org.eclipse.cdt.build.core.buildArtefactType.exe*



Les informations suivantes doivent être ajoutées à l'extension :



org.eclipse.cdt.core.templateAssociations



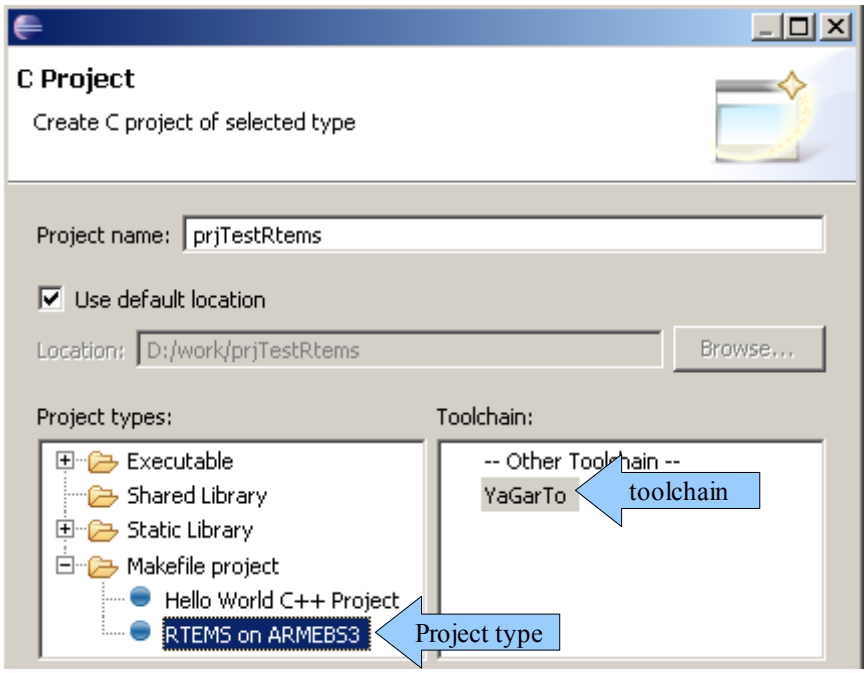
Cela peut paraître paradoxal d'ajouter une toolchain à un projet qui gère les outils utilisés à travers un *makefile* rédigé à la main. Cependant un outils très important mis à disposition par YaGarTo va être réutilisé par un projet de type *RTEMS on ARMEBS3*, il s'agit du debugger. En effet, pour démarrer une session de debuggage il faut connaître le nom de l'outil utilisé (*arm-elf-gdb*), celui-ci est chargé depuis les informations relatives à la toolchain YaGarTo.

Création d'un projet sur Eclipse

Voyons maintenant comment créer un nouveau projet de type *RTEMS on ARMEBS3* à travers Eclipse. Comme expliqué dans la première partie nous avons exporté le plugin sous forme de *.jar* puis ajouté ce dernier à une instance d'Eclipse séparée.

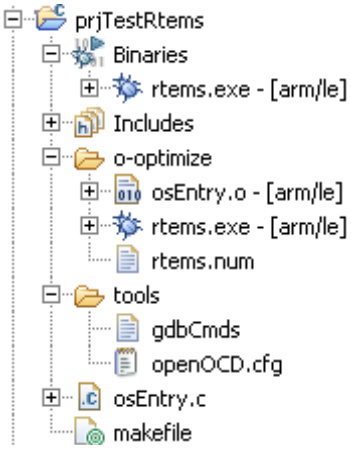
Création du projet

Nous voyons le projet *RTEMS on ARMEBS3* apparaître dans le wizard permettant de créer les nouveaux projets de type C. comme prévu, ce dernier apparaît dans le répertoire *Makefile project*. Il est impératif de sélectionner la toolchain *YaGarTo* pour que le debugger puisse être utilisé.



Lorsque l'on clique sur le bouton *Finish,* comme prévu, les fichiers de base se copient dans le répertoire du projet.

Compilation



Après avoir compilé le projet, on retrouve la même structure que dans les projets de type *Executable* ou *Static Library*. Un dossier *tools* est créé contenant le fichier de configuration d'*OpenOCD* ainsi que le fichier *gdbCmds* regroupant les commandes à envoyer à *gdb*. Lors de la compilation on aperçoit dans la console les flags modifiés plus haut (*-g -O0*).
Apparaît alors l'exécutable représenté par un petit bug car celui-ci contient les informations relatives au debuggage.

Aperçu du fichier source

```
#undef  NDEBUG

#include <bsp.h>
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

rtems_task task1
(rtems_task_argument ignored)
{}

rtems_task Init
(rtems_task_argument ignored)
{}

/* configuration information */
#define CONFIGURE_TEST_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_TEST_NEEDS_CLOCK_DRIVER
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_MAXIMUM_TASKS 3

#define CONFIGURE_INIT

#include <rtems/confdefs.h>
```

Le fichiers source contenant la tâche d'initialisation de l'OS est nommé *osEntry.c*. Ce dernier contient le code de l'utilisateur.

On aperçoit ci-contre les fonctions d'entrées des tâches ainsi que la fonction de la tâche d'initialisation. Pour voir comment utiliser les outils de RTEMS il faut se référer aux annexes 2.*

A la fin du projet on déclare les définitions permettant de spécifier à *RTEMS* les outils utilisés (nombre de tâches, de timers, microseconde par ticks ...) si ces constantes ne sont pas définies, le fichier *confdefs.h* leur assigne une valeur par défaut Ce fichier regroupe toutes les définitions permettant de configurer *RTEMS*. Les annexes 2.* permettent de visualiser un certain nombre de définitions.

Au niveau du makefile

Il ne faut pas oublier de modifier le *makefile* dès lors qu'un outil spécifique est utilisé. Pour ce faire il faut ajouter à la variable *MANAGERS* le nom de l'outil désiré. Voici la liste de la majorité des outils mis à disposition.

```
# optional managers required
MANAGERS=io timer event semaphore message partition
```

Idem pour les fichiers sources, si un fichier est ajouté au projet il faut mettre à jour la variable *CSRCS*

```
# C source names
CSRCS = osEntry.c otherFile.c
COBJS_ = $(CSRCS:.c=.o)
COBJS = $(COBJS_:%=${ARCH}/%)
```

Il est possible d'automatiser la gestion des fichiers sources pour les projets de grande ampleur. La gestion avancée d'un makefile est un art à part entière c'est pourquoi une aide supplémentaire est ajoutée au CD-Rom livré avec ce rapport.

Debugger RTEMS sur Eclipse

La marche à suivre pour lancer le debug d'un exécutable tournant sur *RTEMS* est exactement la même que celle décrite dans la première partie. Pour visualiser les informations *d'OpenOCD* il faut bien entendu ouvrir la console concernée. Un *breakpoint* est placé automatiquement au commencement de la fonction *Init* , une commande a été ajoutée au fichier *gdbCmds*:

```
"/
status = rtems_task_create(
rtems_build_name( 'T', 'A', '1', ' ' ), 1,
RTEMS_MINIMUM_STACK_SIZE, RTEMS_DEFAULT_MODES,
RTEMS_DEFAULT_ATTRIBUTES | RTEMS_FLOATING_POINT, &id);
assert( !status );
```

Pour tester l'exécution des tâches nous allons ajouter deux *breakpoints* à l'intérieur de chaque fonctions associées à une tâche. Nous verrons alors ces tâches se partager le temps processeur :

```
rtems_task task1 (rtems_task_argument
{
    while (1)
    {rtems_task_wake_after(
            RTEMS_YIELD_PROCESSOR) ;
    }
}
```

```
rtems_task task2 (rtems_task_argument
{
    while (1)
    {rtems_task_wake_after(
            RTEMS_YIELD_PROCESSOR) ;
    }
}
```

Si le kernel contient les informations liées au debuggage il est possible d'entrer dans les fonctions de l'exécutif, *rtems_task_wake_after* par exemple. A noter que les sources de l'OS doivent se trouver au même niveau que le dossier contenant le kernel pour pouvoir les afficher lors du debuggage.

Le plugin « Terminal Hevs »

Pour communiquer avec RTEMS nous utilisons un terminal externe à *Eclipse* connecté au port série de l'ordinateur. Pour éviter de devoir passer d'un programme à l'autre (Eclipse et Terminal) un plugin supplémentaire a été développé. Celui-ci se résume à une console permettant de communiquer avec la cible directement depuis Eclipse.

La librairie rxtx v2.1-7r2

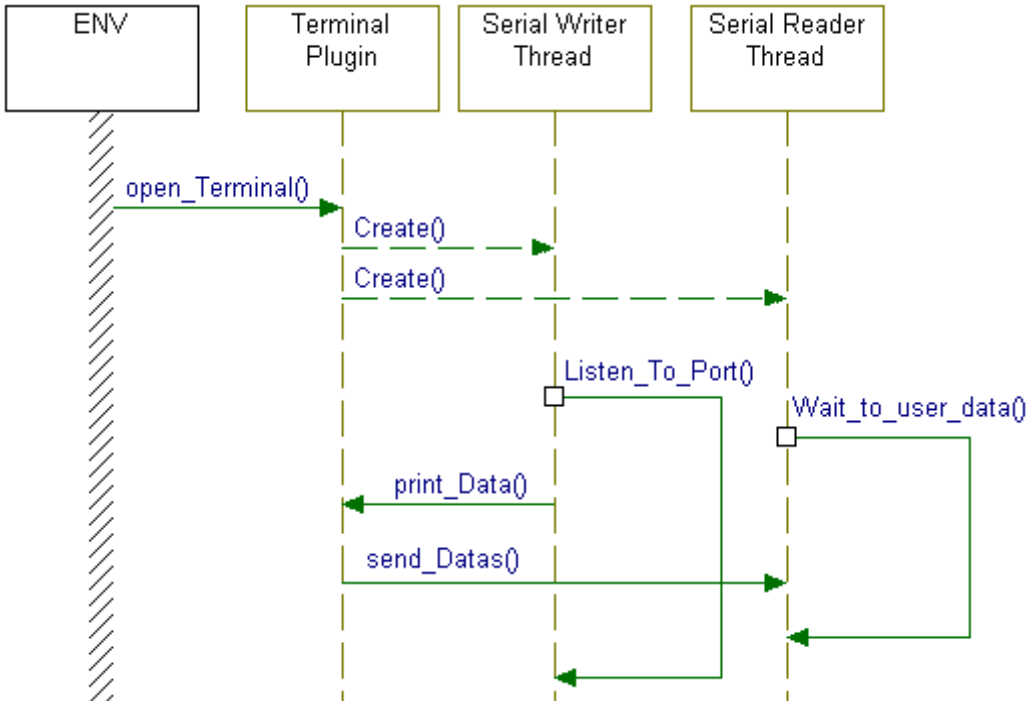
RXTX est une librairie native permettant la communication série et parallèle à travers le JDK. Elle support la librairie standard de communication *CommAPI* de java et met a disposition deux *dll* permettant d'interagir avec le système Windows.

Les extensions utilisées

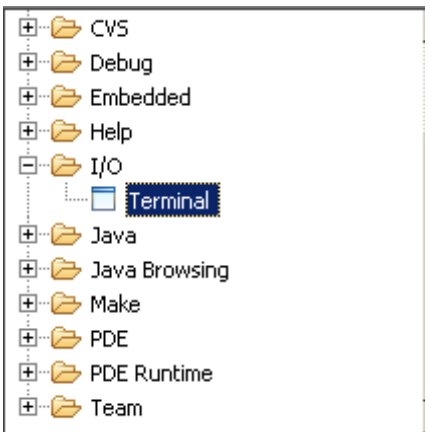
Les extensions utilisées pour communiquer avec la carte sont sensiblement les mêmes que les extensions de la console *OpenOCD*. C'est pourquoi nous allons seulement aborder son utilisation ainsi que le déroulement du code à travers un diagramme de séquence. A noter que la documentation du site explique comment intégrer les *dll* à un plugin Eclipse.

Diagramme de séquence

La gestion de la communication possède deux thread. Le premier permet de récupérer les données envoyées depuis la carte et le deuxième envoie les données du PC vers la cible.

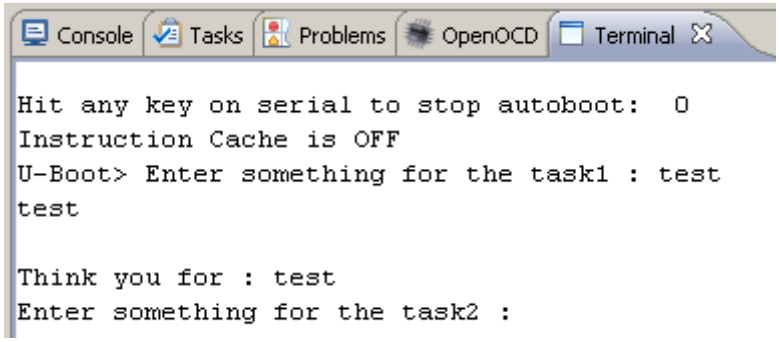


Utilisation du Terminal



Pour ouvrir le terminal il faut cliquer sur le menu *Windows – Show View – Other...* apparaît alors la fenêtre *Show View*, comme pour la vue relative à *OpenOCD*. On constate que la catégorie I/O est correctement créée, elle contient la nouvelle vue intitulée *Terminal*. Pour ouvrir le terminal il suffit de le sélectionner et de cliquer sur OK.

Pour tester notre terminal nous allons ré-ouvrir l'exemple permettant de tester le *printf* et le *scanf* à travers *RTEMS*



Dans l'exemple suivant, les données envoyées à la task1 sont correctement reçue et affichées par la suite. On constate cependant que le *scanf* envoie un écho qui peut devenir pénible si les quantités de données à entrer sont importantes.

Annexe 3.1 TerminalView.java

Améliorations

Le plugin présenté ci-dessus m'a permis de faciliter le test des outils mis à disposition par *RTEMS*. Il serait intéressant de l'améliorer pour que son utilisation soit plus conviviale pour les futurs utilisateurs. Voici les points à améliorer :

- Supprimer l'écho inopiné lors de l'utilisation du *scanf*
- Ajouter une configuration permettant de spécifier le port, le *baudrate*, la taille des datas...
- Enregistrer cette configuration à l'intérieur du projet lui-même. Ainsi les projets nécessitant une configuration spécifique ne devront pas être configurés à chaque fois.

Remerciements

J'aimerais adresser mes remerciements aux personnes suivantes pour leur conseils et leur aide durant la réalisation de ce travail de diplôme.

- Pierre Pompili : Professeur et responsable de la filière *Infotronics*
- Thomas Sterren : Adjoint scientifique
- Marc pignat : Adjoint scientifique
- Samuel Gaist : Adjoint scientifique
- Mes camarades diplômant

Conclusion

Après ces douze semaines je suis satisfait du travail effectué. Je regrette cependant de ne pas avoir eu le temps de tester entièrement l'utilisation du réseau à travers *RTEMS*.

Les outils utilisés durant ce projet m'ont permis d'emmagasiner des connaissances précieuses pour la suite, notamment au niveau d'Eclipse qui tend à s'imposer dans le développement natif ou embarqué. Tout comme RTEMS, qui est un OS réputé dans des milieux à fortes contraintes comme le médical ou le spatial.

Utiliser de l'opensource est une bonne approche pour une école qui n'a pas forcément besoin d'utiliser des produits onéreux présents sur le commerce pour permettre aux étudiant de se familiariser avec le développement embarqué.

Liens

- http://www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html
- http://yagarto.de/
- http://openocd.berlios.de/web/
- http://help.eclipse.org/help32/index.jsp
- http://gcc.gnu.org/onlinedocs/
- http://rtems.com/onlinedocs/releases/rtemsdocs-4.7.1/share/rtems/html/
- http://www.rtems.com/ml/rtems-users/index.html
- http://www.rtems.org/wiki/index.php/Main_Page

Signature

Sion, le 22 novembre 2007 Joël Rochat

Annexes

1 Partie standalone

1.1	crt.s	:	fichier de démarrage d'un standalone
1.2	armebs3_ram.ld	:	fichier de liens d'un standalone
1.3	openOCD.cfg	:	fichier de configuration *d'OpenOCD*
1.4	template.xml	:	instructions exécutées lors de la création d'un nouveau projet de type *Embedded Hevs*
1.5	ManageOpenOCD.java	:	classe permettant de gérer *OpenOCD*

2 Partie RTEMS

2.0	Exemple d'utilisation des fonctions *printf* et *scanf*		
2.1	Code des fonctions *arm_read* et *arm_write*		
2.2	linkcmds	:	fichier de liens du kernel *RTEMS*
2.3	bsp.h	:	fonctions et définitions du BSP utilisé
2.4a	Makefile.inc	:	makefile du kernel *RTEMS*
2.4b	Makefile	:	makefile d'un programme s'exécutant sur *RTEMS*
2.5	Exemple d'utilisation des sémaphores		
2.6	Exemple d'utilisation des queues		
2.7	Exemple d'utilisation des timers		
2.8	Exemple d'utilisation des events		
2.9	Exemple d'utilisation des interruptions		
2.10	Exemple d'utilisation d'une interruption gérée à l'aide d'event		
2.11	Exemple d'utilisation d'une horloge système		
2.12	Exemple d'utilisation de la mémoire dynamiquement (partition)		
2.13	Exemple d'utilisation du réseau		
2.14	Fichier de configuration du réseau		

3 Partie Terminal

3.1	TerminalView.java	:	Classe permettant de gérer un terminal à l'intérieur d'Eclipse

Contenu du CD-ROM

- EclipseIDE : Logiciel de base, avec les plugins nécessaires
- Embedded_Hevs : Sources du plugin *Embedded Hevs*
- Makefile_Help : Aspects principaux des makefiles (pdf)
- OpenOCD : Installateur du programme *OpenOCD*
- Rapport : Rapport au format *pdf* et *odt* et annexes
- Rtems_4.7.1 : Sources de l'OS *RTEMS*
- Rtems_4.7.1_exemples : Exemples officiels
- Rtems_exemples : Exemples rédigés par moi-même mis en annexes
- Rtems_linux_toolchain : Toolchain *arm-rtems4.7* pour Linux (rpms)
- Rtems_windows_toolchain : Toolchain arm-rtems4.7 pour Windows
- Terminal_Hevs : Sources du plugin *Terminal Hevs*
- VMWare : Installateur du logiciel *VMWare*
- *Yagarto* : *Toolchain arm-elf pour standalones.*
- *README.txt* : *Informations diverses et mots de passe de la machine virtuelle Linux.*

Annexes 1

Partie standalone

```
/*** Annexe 1.1 ***/
/*****************************************************************************/
/* FILENAME     :   crt.s                                                    */
/* MODIFIED BY  :   Joël Rochat                                              */
/*---------------------------------------------------------------------------*/
/* FUNCTION     :   C Runtime file                                           */
/*---------------------------------------------------------------------------*/
/* Description  :   Set of execution startup routines required to compile with  */
/*                  GCC                                                      */
/*****************************************************************************/

    /* Some defines for the program status registers*/
   ARM_MODE_USER  = 0x10      /* Normal User Mode                          */
   ARM_MODE_FIQ   = 0x11      /* FIQ Fast Interrupts Mode                  */
   ARM_MODE_IRQ   = 0x12      /* IRQ Standard Interrupts Mode              */
   ARM_MODE_SVC   = 0x13      /* Supervisor Interrupts Mode                */
   ARM_MODE_ABORT = 0x17      /* Abort Processing memory Faults Mode       */
   ARM_MODE_UNDEF = 0x1B      /* Undefined Instructions Mode               */
   ARM_MODE_SYS   = 0x1F      /* System Running in Priviledged Operating Mode */
   ARM_MODE_MASK  = 0x1F

   I_BIT          = 0x80      /* disable IRQ when I bit is set */
   F_BIT          = 0x40      /* disable IRQ when I bit is set */

   .section .vectors,"ax"
   .code 32
/*****************************************************************************/
/*               Vector table and reset entry                                */
/*****************************************************************************/
_vectors:
   ldr pc, ResetAddr    /* Reset                */
   ldr pc, UndefAddr    /* Undefined instruction */
   ldr pc, SWIAddr      /* Software interrupt    */
   ldr pc, PAbortAddr   /* Prefetch abort        */
   ldr pc, DAbortAddr   /* Data abort            */
   ldr pc, ReservedAddr /* Reserved              */
   ldr pc, IRQAddr      /* IRQ interrupt         */
   ldr pc, FIQAddr      /* FIQ interrupt         */

ResetAddr:     .word ResetHandler
UndefAddr:     .word UndefHandler
SWIAddr:       .word SWIHandler
PAbortAddr:    .word PAbortHandler
DAbortAddr:    .word DAbortHandler
ReservedAddr:  .word 0
IRQAddr:       .word IRQHandler
FIQAddr:       .word FIQHandler

   .ltorg

   .section .init, "ax"
   .code 32

   .global ResetHandler
   .global ExitFunction
   .extern main
/*****************************************************************************/
/*                            Reset handler                                  */
/*****************************************************************************/
ResetHandler:
   /*
    * Setup a stack for each mode
    */
   msr   CPSR_c, #ARM_MODE_UNDEF | I_BIT | F_BIT   /* Undefined Instruction Mode */
   ldr   sp, =__stack_und_end
```

```
    msr   CPSR_c, #ARM_MODE_ABORT | I_BIT | F_BIT   /* Abort Mode */
    ldr   sp, =__stack_abt_end

    msr   CPSR_c, #ARM_MODE_FIQ | I_BIT | F_BIT     /* FIQ Mode */
    ldr   sp, =__stack_fiq_end

    msr   CPSR_c, #ARM_MODE_IRQ | I_BIT | F_BIT     /* IRQ Mode */
    ldr   sp, =__stack_irq_end

    msr   CPSR_c, #ARM_MODE_SVC | I_BIT | F_BIT     /* Supervisor Mode */
    ldr   sp, =__stack_svc_end

    /*Clear .bss section*/
    ldr   r1, =__bss_start
    ldr   r2, =__bss_end
    ldr   r3, =0

bss_clear_loop:
    cmp   r1, r2
    strne r3, [r1], #+4
    bne   bss_clear_loop

    /*Jump to main*/

    mrs   r0, cpsr
    bic   r0, r0, #I_BIT | F_BIT     /* Enable FIQ and IRQ interrupt */
    msr   cpsr, r0

    mov   r0, #0 /* No arguments */
    mov   r1, #0 /* No arguments */
    ldr   r2, =main
    mov   lr, pc
    bx    r2     /* And jump... */

ExitFunction:
    nop
    nop
    nop
    b ExitFunction
/*****************************************************************************/
/*                           Default interrupt handler                       */
/*****************************************************************************/
UndefHandler:
    b UndefHandler

SWIHandler:
    b SWIHandler

PAbortHandler:
    b PAbortHandler

DAbortHandler:
    b DAbortHandler

IRQHandler:
    b IRQHandler

FIQHandler:
    b FIQHandler

    .weak ExitFunction
    .weak UndefHandler, PAbortHandler, DAbortHandler
    .weak IRQHandler, FIQHandler
    .ltorg
```

```
/*** Annexe 1.2 ***/
/*****************************************************************************/
/* FILENAME     :   armebs3_ram.ld                                           */
/* MODIFIED BY  :   Joël Rochat                                              */
/*---------------------------------------------------------------------------*/
/* FUNCTION     :   Link File                                                */
/*---------------------------------------------------------------------------*/
/* Description  :   This file place the different parts of exe in the .elf and  */
/*                  define interrupt stack size                              */
/*****************************************************************************/

ENTRY(ResetHandler)
SEARCH_DIR(.)

/*Define stack size here*/
FIQ_STACK_SIZE = 0x0100;
IRQ_STACK_SIZE = 0x0100;
ABT_STACK_SIZE = 0x0100;
UND_STACK_SIZE = 0x0100;
SVC_STACK_SIZE = 0x0400;

MEMORY
{
  ram : org = 0x20000000, len = 128M
}

/*Do not change the next code*/
SECTIONS
{
  .text :
  {
    *(.vectors);
    . = ALIGN(4);
    *(.init);
    . = ALIGN(4);
    *(.text);
    . = ALIGN(4);
    *(.rodata);
    . = ALIGN(4);
    *(.rodata*);
    . = ALIGN(4);
    *(.glue_7t);
    . = ALIGN(4);
    *(.glue_7);
    . = ALIGN(4);
    etext = .;
  } > ram

  .data :
  {
    PROVIDE (__data_start = .);
    *(.data)
    . = ALIGN(4);
    edata = .;
    _edata = .;
    PROVIDE (__data_end = .);
  } > ram

  .bss :
  {
    PROVIDE (__bss_start = .);
    *(.bss)
    *(COMMON)
    . = ALIGN(4);
    PROVIDE (__bss_end = .);
```

```
    . = ALIGN(256);

    PROVIDE (__stack_start = .);

    PROVIDE (__stack_fiq_start = .);
    . += FIQ_STACK_SIZE;
    . = ALIGN(4);
    PROVIDE (__stack_fiq_end = .);

    PROVIDE (__stack_irq_start = .);
    . += IRQ_STACK_SIZE;
    . = ALIGN(4);
    PROVIDE (__stack_irq_end = .);

    PROVIDE (__stack_abt_start = .);
    . += ABT_STACK_SIZE;
    . = ALIGN(4);
    PROVIDE (__stack_abt_end = .);

    PROVIDE (__stack_und_start = .);
    . += UND_STACK_SIZE;
    . = ALIGN(4);
    PROVIDE (__stack_und_end = .);

    PROVIDE (__stack_svc_start = .);
    . += SVC_STACK_SIZE;
    . = ALIGN(4);
    PROVIDE (__stack_svc_end = .);
    PROVIDE (__stack_end = .);
    PROVIDE (__heap_start = .);
  } > ram
}
```

```
/*** Annexe 1.3 ***/
/*****************************************************************************/
/* FILENAME     :   openOCD.cfg                                              */
/* MODIFIED BY  :   Joël Rochat                                              */
/*---------------------------------------------------------------------------*/
/* FUNCTION     :   OpenOCD configuration file                               */
/*---------------------------------------------------------------------------*/
/* Description  :   OpenOCD use this file to configure the connection for our   */
/*                  Target.                                                  */
/*****************************************************************************/

#daemon configuration
telnet_port 4444
gdb_port 3333

#interface
interface ft2232
ft2232_device_desc "Amontec JTAGkey A"
ft2232_layout jtagkey
ft2232_vid_pid 0x0403 0xcff8
jtag_speed 0
jtag_nsrst_delay 200
jtag_ntrst_delay 200

#use combined on interfaces or targets that can't set TRST/SRST separately
reset_config srst_only srst_pulls_trst

#jtag scan chain
#format L IRC IRCM IDCODE (Length, IR Capture, IR Capture Mask, IDCODE)
jtag_device 4 0x1 0xf 0xe

#target configuration
daemon_startup attach

#target <type> <startup mode>
#target arm7tdmi <reset mode> <chainpos> <endianness> <variant>
target arm920t little run_and_halt 0 arm920t
run_and_halt_time 0 30

# For more information about the configuration files, take a look at:
# http://openfacts.berlios.de/index-en.phtml?title=Open+On-Chip+Debugger
```

```
/*** Annexe 1.4 ***/
/*****************************************************************************/
/* FILENAME     :   template.xml                                             */
/* MODIFIED BY  :   Joël Rochat                                              */
/*---------------------------------------------------------------------------*/
/* FUNCTION     :   template file for Embedded ARMEBS3 Projects              */
/*---------------------------------------------------------------------------*/
/* Description  :   template  used to specify default files to load when a new  */
/*                  project is create.                                       */
/*****************************************************************************/
<?xml version="1.0" encoding="ISO-8859-1"?>
<template   type="ProjTempl" version="1.0"
            supplier="Eclipse.org" revision="1.0" author="Joel Rochat"
            copyright=""
            id="ARMEBSDefaultProject" label="Embedded ARMEBS3"
            description="Project for ARMEBS3 development board">

    <process type="org.eclipse.cdt.core.CreateSourceFolder">
        <simple name="projectName" value="$(projectName)"/>
        <simple name="path" value="src"/>
    </process>

    <process type="org.eclipse.cdt.core.AddFiles">
        <simple name="projectName" value="$(projectName)"/>
        <complex-array name="files">
            <element>
                <simple name="source" value="defaultFile.c"/>
                <simple name="target" value="src/$(projectName).c"/>
                <simple name="replaceable" value="true"/>
            </element>
            <element>
                <simple name="source" value="crt.s"/>
                <simple name="target" value="src/crt.s"/>
                <simple name="replaceable" value="true"/>
            </element>
            <element>
                <simple name="source" value="armebs3_ram.ld"/>
                <simple name="target" value="tools/armebs3_ram.ld"/>
                <simple name="replaceable" value="true"/>
            </element>
            <element>
                <simple name="source" value="defaultGdbCmds"/>
                <simple name="target" value="tools/gdbCmds"/>
                <simple name="replaceable" value="true"/>
            </element>
            <element>
                <simple name="source" value="armebs3_openOCD.cfg"/>
                <simple name="target" value="tools/openOCD.cfg"/>
                <simple name="replaceable" value="true"/>
            </element>
        </complex-array>
    </process>
</template>
```

```java
/*** Annexe 1.5 ***/
/*****************************************************************************/
/* FILENAME     :   ManageOpenOCD.java                                       */
/* MODIFIED BY  :   Joël Rochat                                              */
/*---------------------------------------------------------------------------*/
/* FUNCTION     :   This classe is used to Start and Stop OpenOCD. It display   */
/*                  Informations from OpenOCD in the console and Kill it when   */
/*                  Eclipse exit                                             */
/*****************************************************************************/

package hevs.embedded.tools;

import hevs.embedded.view.OpenOCDConsole;

public class ManageOpenOCD
{
    private Process exe = null ;
    private  String openOCDCmd = null ;
    private  String openOCDConfigFile = null ;

    //manage the openOCD display
    private ManagePrint threadPrinter = null ;

    private static ManageOpenOCD openOCD = null ;

    public ManageOpenOCD ()
    {
        if (openOCD == null)
        {
            openOCD = this ;

            //we had the thread to the Runtime
            Runtime.getRuntime().addShutdownHook(new KillOpenOCD()) ;

            System.out.println("Manager create") ;
        }
    }

    //return a pointer on the current class
    public static ManageOpenOCD getDefault()
    {
        return openOCD ;
    }

    //give the path of openocd-ftd2xx and the path of the configuration file
    public void setOpenOCDInfo (String cmdPath, String configFile)
    {
        openOCDCmd = cmdPath ;
        openOCDConfigFile = configFile ;
    }

    public void restart ()
    {
        if (openOCD != null && openOCDConfigFile != null)
        {
            try
            {
                kill () ;

                ProcessBuilder builder = new ProcessBuilder
                                        (openOCDCmd, "-f", openOCDConfigFile) ;
                //C:/openocd/openocd-ftd2xx -f D:/OpenOCD.cfg

                exe = builder.start() ;
                threadPrinter = new ManagePrint() ;
                threadPrinter.start() ;
```

```
        }
        catch (Exception e) {System.out.println("error") ;}
    }
}

public void kill ()
{
    if (exe != null)
    {
        exe.destroy() ;

        try
        {
            //wait for exe death
            exe.waitFor() ;
        }
        catch (InterruptedException e) {}

        exe = null ;
    }

    if (threadPrinter != null)
    {
        threadPrinter.halt() ;

        //wait for the halt of the thread
        while (threadPrinter.isAlive()){}

        threadPrinter = null ;
    }
}
//This Thread manage the display of OpenOCD informations
public class ManagePrint extends Thread
{
    private boolean stopThread = false;
    private BufferedReader infoReader = null ;

    public void run ()
    {
        boolean fin = false;
        String currentLine = null ;
        infoReader = new BufferedReader(
                new InputStreamReader(exe.getErrorStream()));
        try
        {
            OpenOCDConsole.getDefault().clear() ;

            while (!fin)
            {
                //the infoReader.readLine() method block the Thread
                //we must destroy the openOCD process before halt this Thread
                while((currentLine = infoReader.readLine()) != null)
                    OpenOCDConsole.getDefault().print(currentLine) ;

                //make this thread stops correctly
                synchronized(this)
                {
                    fin = this.stopThread ;
                }
            }
        }
        catch (Exception e) {System.out.println(e.getMessage());}
    }

    public synchronized void halt()
    {
```

```
            this.stopThread = true ;
            //close the stream from openocd
            try{infoReader.close();}catch (Exception e){
                System.out.println("unable to close") ;}
        }
    }
    //this class kills openOCD and the Thread when eclipse stops
    public class KillOpenOCD extends Thread
    {
        public void run()
        {
            if (exe != null)
                exe.destroy() ;

            if (threadPrinter != null)
                threadPrinter.halt() ;

            System.out.println("Threads killed") ;
        }
    }

}
```

Annexes 2

Partie RTEMS

```c
/*** Annexe 2.0 ***/
/*****************************************************************************/
/* FILENAME     :   osEntry.c                                                */
/* AUTHOR       :   Rochat Joel                                              */
/*---------------------------------------------------------------------------*/
/* FUNCTION     :   Test of arm_write and arm_read functions                  */
/*---------------------------------------------------------------------------*/
/* Description  :   The tasks read some text the user write and display it again*/
/*                  The tasks have the same priority, they work in round robin  */
/*                  mode.                                                    */
/*****************************************************************************/
#undef  NDEBUG

#include <bsp.h>
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

/*taks 1 function*/
rtems_task task1 (rtems_task_argument ignored)
{
    char txt[20] ;
    for(;;)
    {
        /*print the text to Terminal*/
        printf ("Enter something for the task1 : ") ;
        fflush (stdout) ;

        /*wait for text*/
        scanf ("%s", txt) ;

        /*print the text received*/
        printf ("\n") ;
        printf ("Think you for : %s\n", txt) ;
        fflush (stdout) ;

        /*sleep*/
        rtems_task_wake_after(RTEMS_YIELD_PROCESSOR) ;
    }
}

/*taks 1 function*/
rtems_task task2 (rtems_task_argument ignored)
{
    char txt[20] ;
    for(;;)
    {
        /*print the text to Terminal*/
        printf ("Enter something for the task2 : ") ;
        fflush (stdout) ;

        /*wait for text*/
        scanf ("%s", txt) ;

        /*print the text received*/
        printf ("\n") ;
        printf ("Think you for : %s\n", txt) ;
        fflush (stdout) ;

        /*sleep*/
        rtems_task_wake_after(RTEMS_YIELD_PROCESSOR) ;
    }
}

/*Task init*/
```

```c
rtems_task Init (rtems_task_argument ignored)
{
    rtems_status_code status;
    rtems_id          id;

    /*Create and start the first task*/
    status = rtems_task_create(
        rtems_build_name( 'T', 'A', '1', ' ' ), 1,
        RTEMS_MINIMUM_STACK_SIZE, RTEMS_DEFAULT_MODES,
        RTEMS_DEFAULT_ATTRIBUTES | RTEMS_FLOATING_POINT, &id);
    assert( !status );

    status = rtems_task_start( id, task1, 0 );
    assert( !status );

    /*Create and start the second task*/
    status = rtems_task_create(
        rtems_build_name( 'T', 'A', '2', ' ' ), 1,
        RTEMS_MINIMUM_STACK_SIZE, RTEMS_DEFAULT_MODES,
        RTEMS_DEFAULT_ATTRIBUTES | RTEMS_FLOATING_POINT, &id);
    assert( !status );

    status = rtems_task_start( id, task2, 0 );
    assert( !status );

    status = rtems_task_delete( RTEMS_SELF );

    exit( 0 );
}
/* configuration informations */
#define CONFIGURE_TEST_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_TEST_NEEDS_CLOCK_DRIVER
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_MAXIMUM_TASKS 3
#define CONFIGURE_INIT
#include <rtems/confdefs.h>
```

```
/*** Annexe 2.1 ***/
/*****************************************************************************/
/* FUNCTION:    read                                                         */
/* INPUTS  :    - fno   -> type of transfer (0 for a read access)            */
/*              - *buf  -> address of text buffer                            */
/*              - size  -> number of chars to write                          */
/* OUTPUTS :    returns number of readed char (always 0)                     */
/*---------------------------------------------------------------------------*/
/* COMMENTS:    Overloads the standard I/O Multi2000 write output            */
/*****************************************************************************/
INT32 arm_read(INT32 fno, const void *buf, INT32 size)
{
    char * msgPtr = (char *)buf;                /* local buffer pointer         */
    /* wait the RS232 receiver has a char in his register                       */
    while((AT91C_BASE_US0->US_CSR & AT91C_US_RXRDY)==0){};
    *msgPtr = AT91C_BASE_US0->US_RHR;           /* read received data           */
    arm_write(1, msgPtr, 1) ;
    return 1;                                   /* return one char readed       */
}


/*****************************************************************************/
/* FUNCTION:    write                                                        */
/* INPUTS  :    - fno   -> type of transfer (1 for a write access)           */
/*              - *buf  -> address of text buffer                            */
/*              - size  -> number of chars to write                          */
/* OUTPUTS :    returns 0 if no error occurs                                 */
/*---------------------------------------------------------------------------*/
/* COMMENTS:    Overloads the standard I/O RTEMS write output                */
/*****************************************************************************/
INT32 arm_write(INT32 fno, const void *buf, INT32 size)
{
    INT32 i;        /* local counter*/
    char * msgPtr = (char *)buf;    /* local buffer pointer */

    for(i=0;i<size;i++)/* for the size of message*/
    {
        if((*msgPtr)=='\n')/* for CR-LF send automatic*/
        {
            while((AT91C_BASE_US0->US_CSR & AT91C_US_TXRDY)==0){};
            AT91C_BASE_US0->US_THR = '\r';/* send LF char on RS232*/

        }
        /* wait the RS232 transmitterisready*/
        while((AT91C_BASE_US0->US_CSR & AT91C_US_TXRDY)==0){};
            AT91C_BASE_US0->US_THR = *msgPtr++; /* send one char on RS232*/
    }
    return 0;                                   /* returns no error */
}
```

```
/*** Annexe 2.2 ***/
/*****************************************************************************/
/* FILENAME     :   linkcmds                                                 */
/* AUTHOR       :   Cogent Computer Systems                                  */
/*---------------------------------------------------------------------------*/
/* FUNCTION     :   Organise ram size, irq stack size and data sections      */
/*---------------------------------------------------------------------------*/
/* Description  :   The equivalent file as armebs3.ld file for
standalones     */                                                    */
/*****************************************************************************/

OUTPUT_FORMAT("elf32-littlearm", "elf32-bigarm",
          "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)

MEMORY
{
    sdram : ORIGIN = 0x20100000, LENGTH = 15M
    sram  : ORIGIN = 0x00200000, LENGTH = 16K
}

/*
 * Declare some sizes.
 */
_sdram_base = DEFINED(_sdram_base) ? _sdram_base : 0x20100000;
_sdram_size = DEFINED(_sdram_size) ? _sdram_size : 15M;


_sram_base = DEFINED(_sram_base) ? _sram_base : 0x00000000;
_sram_size = DEFINED(_sram_size) ? _sram_size : 16K;

_irq_stack_size = DEFINED(_irq_stack_size) ? _irq_stack_size : 0x1000;
_fiq_stack_size = DEFINED(_fiq_stack_size) ? _fiq_stack_size : 0x400;
_abt_stack_size = DEFINED(_abt_stack_size) ? _abt_stack_size : 0x100;
_svc_stack_size = DEFINED(_svc_stack_size) ? _svc_stack_size : 0x1000;



/* Do we need any of these for elf?
   __DYNAMIC = 0;    */

SECTIONS
{
    .base :
    {
        _sram_base = .;

    /* reserve room for the vectors and function pointers */
    arm_exception_table = .;
    . += 64;

    /* 256 byte aligned rx buffer header array */
    . = ALIGN (0x100);
    at91rm9200_emac_rxbuf_hdrs = .;

    /* 1 transmit buffer, 0x600 size */
    . += (0x100);
    at91rm9200_emac_txbuf = .;
    . += (0x600);

    /* 8 receive buffers, 0x600 each */
    at91rm9200_emac_rxbufs = .;
    . += (0x600 * 8);

    } > sram
```

```
.init          :
{
    KEEP (*(.init))
} > sdram   /*=0*/

.text      :
{
_text_start = .;
 CREATE_OBJECT_SYMBOLS
*(.text)
*(.text.*)

    /*
     * Special FreeBSD sysctl sections.
     */
    . = ALIGN (16);
    __start_set_sysctl_set = .;
    *(set_sysctl_*);
    __stop_set_sysctl_set = ABSOLUTE(.);
    *(set_domain_*);
    *(set_pseudo_*);

/* .gnu.warning sections are handled specially by elf32.em.  */
*(.gnu.warning)
*(.gnu.linkonce.t*)
*(.glue_7)
*(.glue_7t)

/* I think these come from the ld docs: */
___CTOR_LIST__ = .;
LONG((___CTOR_END__ - ___CTOR_LIST__) / 4 - 2)
*(.ctors)
LONG(0)
___CTOR_END__ = .;
___DTOR_LIST__ = .;
LONG((___DTOR_END__ - ___DTOR_LIST__) / 4 - 2)
*(.dtors)
LONG(0)
___DTOR_END__ = .;

_etext = .;
PROVIDE (etext = .);
} > sdram

.fini      :
{
    KEEP (*(.fini))
} > sdram  /*=0*/

.data :
{
    *(.data)
    *(.data.*)
    *(.gnu.linkonce.d*)
    *(.jcr)
    SORT(CONSTRUCTORS)
    _edata = .;
} > sdram

.eh_frame : { *(.eh_frame) } > sdram
.data1   : { *(.data1) } > sdram
.eh_frame : { *(.eh_frame) } > sdram
.gcc_except_table : { *(.gcc_except_table) } > sdram

.rodata :
```

```
    {
      *(.rodata)
      *(.rodata.*)
      *(.gnu.linkonce.r*)
    } > sdram

    .bss       :
    {
    _bss_start_ = .;
    _clear_start = .;
    *(.bss)
    *(.bss.*)
    *(.gnu.linkonce.b.*)
    *(COMMON)
    . = ALIGN(64);
    _clear_end = .;

    . = ALIGN (256);
    _abt_stack = .;
    . += _abt_stack_size;

    . = ALIGN (256);
    _irq_stack = .;
    . += _irq_stack_size;

    . = ALIGN (256);
    _fiq_stack = .;
    . += _fiq_stack_size;

    . = ALIGN (256);
    _svc_stack = .;
    . += _svc_stack_size;

    _bss_end_ = .;
    _end = .;
    __end = .;

/*
 * Ideally, the MMU's translation table would be in SRAM. But we need
 * 16K which is the size of SRAM. If we do the mapping right, the TLB
 * should be big enough that to hold all the translations that matter,
 * so keeping the table in SDRAM won't be a problem.
 */
    . = ALIGN (16 * 1024);
      _ttbl_base = .;
      . += (16 * 1024);


    . = ALIGN (1024);
    _bss_free_start = .;

    } > sdram


/* Debugging stuff follows? */

    /* Stabs debugging sections.  */
    .stab 0 : { *(.stab) }
    .stabstr 0 : { *(.stabstr) }
    .stab.excl 0 : { *(.stab.excl) }
    .stab.exclstr 0 : { *(.stab.exclstr) }
    .stab.index 0 : { *(.stab.index) }
    .stab.indexstr 0 : { *(.stab.indexstr) }
    .comment 0 : { *(.comment) }
    /* DWARF debug sections.
       Symbols in the DWARF debugging sections are relative to the beginning
```

```
       of the section so we begin them at 0.  */
    /* DWARF 1 */
    .debug          0 : { *(.debug) }
    .line           0 : { *(.line) }
    /* GNU DWARF 1 extensions */
    .debug_srcinfo  0 : { *(.debug_srcinfo) }
    .debug_sfnames  0 : { *(.debug_sfnames) }
    /* DWARF 1.1 and DWARF 2 */
    .debug_aranges  0 : { *(.debug_aranges) }
    .debug_pubnames 0 : { *(.debug_pubnames) }
    /* DWARF 2 */
    .debug_info     0 : { *(.debug_info) }
    .debug_abbrev   0 : { *(.debug_abbrev) }
    .debug_line     0 : { *(.debug_line) }
    .debug_frame    0 : { *(.debug_frame) }
    .debug_str      0 : { *(.debug_str) }
    .debug_loc      0 : { *(.debug_loc) }
    .debug_macinfo  0 : { *(.debug_macinfo) }
    /* SGI/MIPS DWARF 2 extensions */
    .debug_weaknames 0 : { *(.debug_weaknames) }
    .debug_funcnames 0 : { *(.debug_funcnames) }
    .debug_typenames 0 : { *(.debug_typenames) }
    .debug_varnames  0 : { *(.debug_varnames) }
/*    .stack 0x80000 : { _stack = .; *(.stack) }*/
  /* These must appear regardless of  .  */
}
```

```
/*** Annexe 2.3 ***/
/*****************************************************************************/
/* FILENAME     :   bsp.h                                                    */
/* AUTHOR       :   Cogent Computer Systems                                  */
/*---------------------------------------------------------------------------*/
/* Description  :   Specific functions and definition for the bsp used : csb337 */
/*****************************************************************************/

#ifndef _BSP_H
#define _BSP_H

#ifdef __cplusplus
extern "C" {
#endif

#include <bspopts.h>

#include <rtems.h>
#include <rtems/console.h>
#include <rtems/clockdrv.h>
#include <libchip/serial.h>


/* What is the input clock freq in hertz? */
#define BSP_MAIN_FREQ 20000000      /* 20.0 MHz */
#define BSP_SLCK_FREQ   32768      /* 32.768 KHz */

/* What is the last interrupt? */
#define BSP_MAX_INT AT91RM9200_MAX_INT

console_tbl *BSP_get_uart_from_minor(int minor);
static inline int32_t BSP_get_baud(void) {return 38400;}

/* How many serial ports? */
#define CONFIGURE_NUMBER_OF_TERMIOS_PORTS 1

/* How big should the interrupt stack be? */
#define CONFIGURE_INTERRUPT_STACK_MEMORY  (16 * 1024)

extern rtems_configuration_table BSP_Configuration;

#define ST_PIMR_PIV 33  /* 33 ticks of the 32.768Khz clock ~= 1msec

/*
 * Network driver configuration
 */
extern struct rtems_bsdnet_ifconfig *config;

/* Change these to match your board */
int rtems_at91rm9200_emac_attach(struct rtems_bsdnet_ifconfig *config, int attaching);
#define RTEMS_BSP_NETWORK_DRIVER_NAME   "eth0"
#define RTEMS_BSP_NETWORK_DRIVER_ATTACH rtems_at91rm9200_emac_attach

#ifdef __cplusplus
}
#endif

#endif /* _BSP_H */
```

```
//*** Annexe 2.4 ***/
/*****************************************************************************/
/* FILENAME     :   Makefile.inc                                             */
/* AUTHOR       :   Cogent Computer Systems                                  */
/*---------------------------------------------------------------------------*/
/* Description  :   Makefile of RTEMS kernel, used to merge                  */
/*                  OS and user's code                                       */
/*****************************************************************************/

RTEMS_BSP = csb337

prefix = D:/rtemsForArm/
exec_prefix = D:/rtemsForArm/arm-rtems4.7

CC_FOR_TARGET = arm-rtems4.7-gcc --pipe
AS_FOR_TARGET = arm-rtems4.7-as
AR_FOR_TARGET = arm-rtems4.7-ar
NM_FOR_TARGET = arm-rtems4.7-nm
LD_FOR_TARGET = arm-rtems4.7-ld
SIZE_FOR_TARGET = arm-rtems4.7-size
OBJCOPY_FOR_TARGET = arm-rtems4.7-objcopy

CC= $(CC_FOR_TARGET)
AS= $(AS_FOR_TARGET)
LD= $(LD_FOR_TARGET)
NM= $(NM_FOR_TARGET)
AR= $(AR_FOR_TARGET)
SIZE= $(SIZE_FOR_TARGET)
OBJCOPY= $(OBJCOPY_FOR_TARGET)

export CC
export AS
export LD
export NM
export AR
export SIZE
export OBJCOPY

RTEMS_CUSTOM = $(prefix)/make/custom/$(RTEMS_BSP).cfg
PROJECT_ROOT = $(prefix)
RTEMS_USE_OWN_PDIR = no
RTEMS_HAS_POSIX_API = no
RTEMS_HAS_ITRON_API = yes
RTEMS_HAS_CPLUSPLUS = no

export RTEMS_BSP
export RTEMS_CUSTOM
export PROJECT_ROOT

# FIXME: The following shouldn't be here
RTEMS_ROOT = $(PROJECT_ROOT)
export RTEMS_ROOT
```

```
//*** Annexe 2.4 ***/
/*****************************************************************************/
/* FILENAME     :   Makefile                                                 */
/* AUTHOR       :   Cogent Computer Systems                                  */
/*---------------------------------------------------------------------------*/
/* Description  :   Makefile of an RTEMS users's code                        */
/*****************************************************************************/
#
#  Makefile
#

#
#  RTEMS_MAKEFILE_PATH is typically set in an environment variable
#

EXEC=test.exe
PGM=${ARCH}/$(EXEC)

# optional managers required
MANAGERS=io timer event

# C source names
CSRCS = osEntry.c
COBJS_ = $(CSRCS:.c=.o)
COBJS = $(COBJS_:%=${ARCH}/%)

# C++ source names
CXXSRCS =
CXXOBJS_ = $(CXXSRCS:.cc=.o)
CXXOBJS = $(CXXOBJS_:%=${ARCH}/%)

# AS source names
ASSRCS =
ASOBJS_ = $(ASSRCS:.s=.o)
ASOBJS = $(ASOBJS_:%=${ARCH}/%)

# Libraries
LIBS = -lrtemsall -lc

include $(RTEMS_MAKEFILE_PATH)/Makefile.inc

include $(RTEMS_CUSTOM)
include $(PROJECT_ROOT)/make/leaf.cfg

OBJS= $(COBJS) $(CXXOBJS) $(ASOBJS)

all:    ${ARCH} $(PGM)

$(PGM): $(OBJS)
    $(make-exe)
```

```
/*** Annexe 2.5 ***/
/*****************************************************************************/
/* FILENAME     :   osEntry.c                                                */
/* AUTHOR       :   Rochat Joel                                              */
/*---------------------------------------------------------------------------*/
/* FUNCTION     :   Test of semaphores/mutex functions of rtems              */
/*---------------------------------------------------------------------------*/
/* Description  :   The task print there name char per char using a mutex.   */
/*                  The tasks share the standard output. If you try to delete*/
/*                  the code who manage the mutex the output will be a mess  */
/*****************************************************************************/
#undef  NDEBUG

#include <bsp.h>
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*The mutex id is passed as argument*/
rtems_task task1 (rtems_id mutex)
{
    char *txt = "Task one is speaking" ;
    int i = 0 ;

    while (1)
    {
        //The task want to display her first char, we make a P()
        //If the standard output is already used we wait for it to be release
        if (i == 0)
            rtems_semaphore_obtain(mutex, RTEMS_WAIT, RTEMS_NO_TIMEOUT) ;

        //Print the char
        printf ("%c", txt[i++]) ;

        //It's the last char, we make V()
        if (i == strlen(txt))
        {
            i = 0 ;
            printf ("\n") ;
            rtems_semaphore_release(mutex) ;
        }

        fflush (stdout) ;

        //sleep
        rtems_task_wake_after(10) ;
    }
}

/*The mutex id is passed as argument*/
rtems_task task2 (rtems_id mutex)
{
    char *txt = "Task two is speaking" ;
    int i = 0 ;

    while (1)
    {
        //The task want to display her first char, we make a P()
        //If the standard output is already used we wait for it to be release
        if (i == 0)
            rtems_semaphore_obtain(mutex, RTEMS_WAIT, RTEMS_NO_TIMEOUT) ;

        //Print the char
        printf ("%c", txt[i++]) ;
```

```
        //It's the last char, we make V()
        if (i == strlen(txt))
        {
            i = 0 ;
            printf ("\n") ;
            rtems_semaphore_release(mutex) ;
        }

        fflush (stdout) ;

        //sleep
        rtems_task_wake_after(10) ;
    }
}

rtems_task Init (rtems_task_argument ignored)
{
    rtems_status_code status ;
    rtems_id id ;
    rtems_id semId ;

    /*Creat the semaphore*/
    status = rtems_semaphore_create (
            rtems_build_name( 'S', 'E', '1', ' ' ), 1, //mutex = 1 place
            RTEMS_DEFAULT_ATTRIBUTES,
            1,
            &semId);

    assert( !status );

    /*Create and start the first task*/
    status = rtems_task_create(
        rtems_build_name( 'T', 'A', '1', ' ' ), 1,
        RTEMS_MINIMUM_STACK_SIZE, RTEMS_DEFAULT_MODES,
        RTEMS_DEFAULT_ATTRIBUTES | RTEMS_FLOATING_POINT, &id);
    assert( !status );

    //the mutex id is givent as argument
    status = rtems_task_start( id, task1, semId );
    assert( !status );

    /*Create and start the second task*/
    status = rtems_task_create(
        rtems_build_name( 'T', 'A', '2', ' ' ), 1,
        RTEMS_MINIMUM_STACK_SIZE, RTEMS_DEFAULT_MODES,
        RTEMS_DEFAULT_ATTRIBUTES | RTEMS_FLOATING_POINT, &id);
    assert( !status );

    //the mutex id is givent as argument
    status = rtems_task_start( id, task2, semId );
    assert( !status );

    status = rtems_task_delete( RTEMS_SELF );

    exit( 0 );
}

/* configuration information */
#define CONFIGURE_TEST_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_TEST_NEEDS_CLOCK_DRIVER
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_MAXIMUM_TASKS 3
#define CONFIGURE_INIT

#include <rtems/confdefs.h>
```

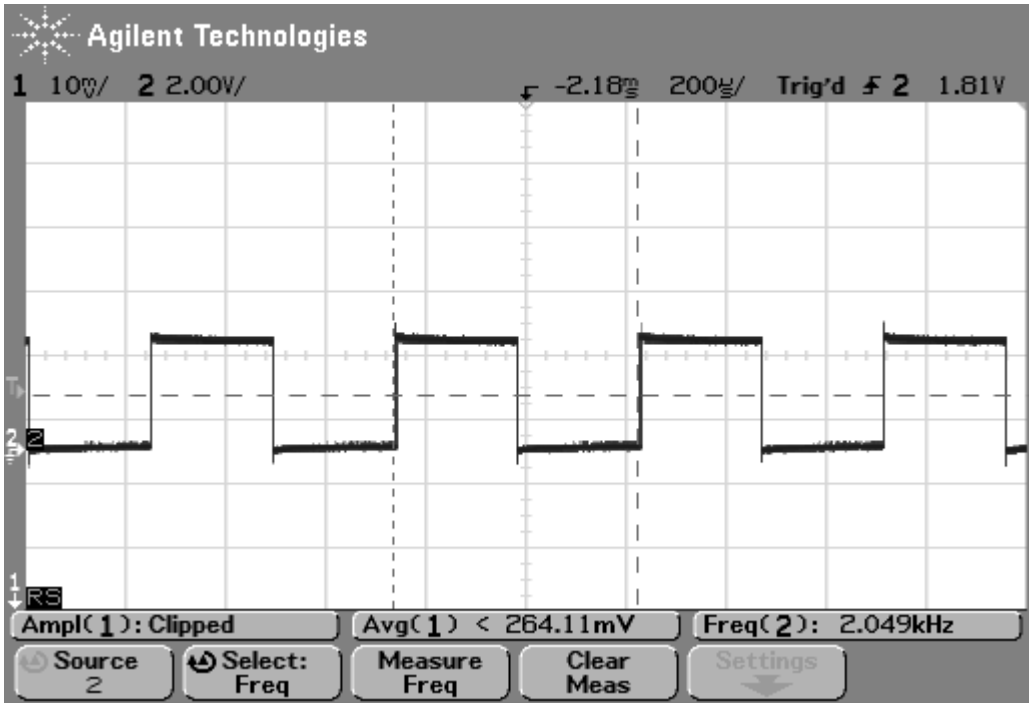
```c
/*** Annexe 2.6 ***/
/*****************************************************************************/
/* FILENAME     :   osEntry.c                                                */
/* AUTHOR       :   Rochat Joel                                              */
/*---------------------------------------------------------------------------*/
/* FUNCTION     :   Test of message queue                                    */
/*---------------------------------------------------------------------------*/
/* Description  :   The task1 and task2 don't work at the same speed. A message */
/*                  queue is used to synchronize the two threads. Infos are  */
/*                  displayed to see when a queue is full or empty, you can play*/
/*                  with the sleeping time of the threads to see how it works   */
/*****************************************************************************/
#undef  NDEBUG

#include <bsp.h>
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <rtems/error.h>

/*message size in bytes 4 bytes for a 32 bits Integer*/
#define MESSAGE_SIZE 4

/*how long the sender sleeps*/
#define SEND_SLEEP 20

/*how long the receiver sleeps*/
#define RECEIVE_SLEEP 30

rtems_task send(rtems_id queue)
{
    int i = 1 ;
    rtems_status_code status ;

    while (1)
    {
        /*send counter to the queue*/
        status = rtems_message_queue_send(queue, (int*)&i, MESSAGE_SIZE);

        /*check if the message was successfully send*/
        if (status != RTEMS_SUCCESSFUL)
        {
            printf ("It's full !\n") ;
            fflush(stdout) ;
        }
        else
        {   /*The message was send, we increment the
                counter for the next message*/
            i++ ;
        }

        rtems_task_wake_after(SEND_SLEEP) ;
    }
}

rtems_task receive(rtems_id queue)
{
    int msgReceived ;

    /*to disable warnings*/
    rtems_unsigned32 size = MESSAGE_SIZE ;
    rtems_status_code status ;

    while (1)
    {
```

```
        status = rtems_message_queue_receive(
                queue, (int*)(&msgReceived),
                &size, RTEMS_NO_WAIT, RTEMS_NO_TIMEOUT);

        if (status == RTEMS_SUCCESSFUL)
            printf ("Received : %d\n", msgReceived) ;
        else
            printf ("It's empty !\n") ;

        fflush(stdout) ;

        rtems_task_wake_after(RECEIVE_SLEEP) ;
    }
}

rtems_task Init(rtems_task_argument ignored)
{
    rtems_status_code status;
    rtems_id queueId ;
    rtems_id id ;

    /*create the queue*/
    status = rtems_message_queue_create(
        rtems_build_name( 'Q', 'U', '1', ' ' ),
        5, //number of place
        MESSAGE_SIZE, //size of place
        RTEMS_DEFAULT_ATTRIBUTES,
        &queueId);
    assert( !status );

    /*Create and start the send task*/
    status = rtems_task_create(
        rtems_build_name( 'T', 'A', '1', ' ' ), 1,
        RTEMS_MINIMUM_STACK_SIZE, RTEMS_DEFAULT_MODES,
        RTEMS_DEFAULT_ATTRIBUTES | RTEMS_FLOATING_POINT, &id);
    assert( !status );

    /*the id of the queue is passed as argument*/
    status = rtems_task_start( id, send, queueId );
    assert( !status );

    /*Create and start the receive task*/
    status = rtems_task_create(
        rtems_build_name( 'T', 'A', '1', ' ' ), 1,
        RTEMS_MINIMUM_STACK_SIZE, RTEMS_DEFAULT_MODES,
        RTEMS_DEFAULT_ATTRIBUTES | RTEMS_FLOATING_POINT, &id);
    assert( !status );

    /*the id of the queue is passed as argument*/
    status = rtems_task_start( id, receive , queueId );
    assert( !status );

    status = rtems_task_delete( RTEMS_SELF );
    exit( 0 );
}

/* configuration information */
#define CONFIGURE_TEST_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_TEST_NEEDS_CLOCK_DRIVER
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_MAXIMUM_TASKS 4
#define CONFIGURE_MAXIMUM_MESSAGE_QUEUES 1
#define CONFIGURE_INIT

#include <rtems/confdefs.h>
```

```
/*** Annexe 2.7 ***/
/*****************************************************************************/
/* FILENAME       :     osEntry.c                                            */
/* AUTHOR         :     Rochat Joel                                          */
/*---------------------------------------------------------------------------*/
/* FUNCTION       :     Test of rtems timer                                  */
/*---------------------------------------------------------------------------*/
/* Description    :     The task displayTime display a chrono. A timer is    */
/*                      used to know when the task have to increment the     */
/*                      timer                                                */
/*****************************************************************************/

#undef  NDEBUG

#include <bsp.h>
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

/*variable to know when we
 *have to increment the time*/
int gTick ;

/*global second counter*/
int sec ;

/*function called by the timer*/
rtems_timer_service_routine timer1Hz (rtems_id timer_id, void *user_data)
{
      gTick = TRUE ;
      rtems_timer_reset(timer_id) ;
}

/*function to display the chrono*/
rtems_task displayTime (rtems_task_argument ignored)
{
      while (1)
      {
            if (gTick)
            {
                  /*display time like 00:01:07*/
                  sec++ ;
                  printf("\r%02d:%02d:%02d", sec/3600, (sec%3600)/60, sec%60) ;
                  fflush(stdout) ;
                  gTick = FALSE ;
            }
      }
}

/*task to Init the system*/
rtems_task Init(rtems_task_argument ignored)
{
      rtems_status_code status;
      rtems_id timerId ;
      rtems_id id ;

      /*Create the timer*/
      status = rtems_timer_create (
                  rtems_build_name( 'T', 'A', '1', ' ' ),
                  &timerId);
      assert(!status ) ;

      /*Create the task*/
```

```
        status = rtems_task_create(
                    rtems_build_name( 'T', 'A', '1', ' ' ), 1,
                    RTEMS_MINIMUM_STACK_SIZE, RTEMS_DEFAULT_MODES,
                    RTEMS_DEFAULT_ATTRIBUTES | RTEMS_FLOATING_POINT, &id);
        assert( !status );

        gTick = FALSE ;
        sec = 0 ;

        /* Start the timer, each 100 ticks
         * the function timer1Hz is called
         * 100 ticks * 10 ms per ticks = 1 s
         */
        status = rtems_timer_fire_after(timerId, 100, timer1Hz, NULL);
        assert( !status );

        /*Start the task*/
        status = rtems_task_start( id, displayTime, 0);
        assert( !status );

        status = rtems_task_delete( RTEMS_SELF );

        exit( 0 );
}

/* configuration information */
/*There is 10ms per ticks = 10'000us*/
#define CONFIGURE_MICROSECONDS_PER_TICK RTEMS_MILLISECONDS_TO_MICROSECONDS(10)
#define CONFIGURE_MAXIMUM_TIMERS 1
#define CONFIGURE_MAXIMUM_TASKS 2
#define CONFIGURE_TEST_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_TEST_NEEDS_CLOCK_DRIVER
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_INIT

#include <rtems/confdefs.h>
```

Here is the mesure of the 2kHz timer :



```c
/*** Annexe 2.8 ***/
/*****************************************************************************/
/* FILENAME     :   osEntry.c                                                */
/* AUTHOR       :   Rochat Joel                                              */
/*---------------------------------------------------------------------------*/
/* FUNCTION     :   Test of event directives                                 */
/*---------------------------------------------------------------------------*/
/* Description  :   A Timer send event 8 and event 7 one after the other. The   */
/*                  task1 is waiting on events and detect wich event occure and */
/*                  write infos to standard output. There is an event set of 32 */
/*                  events between RTEMS_EVENT_0 and RTEMS_EVENT_31              */
/*****************************************************************************/

#undef  NDEBUG

#include <bsp.h>
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>
#include <string.h>

/*use to alternate
 * between event 7 and 8*/
int flag ;

/*function called by the timer*/
rtems_timer_service_routine timer1Hz (rtems_id timer_id, void *user_data)
{
    rtems_id taskId ;

    /*the task id is given as argument*/
    taskId = *((rtems_id*)(user_data)) ;

    /*alternate between events*/
    if (flag)
        rtems_event_send(taskId, RTEMS_EVENT_8) ;
    else
        rtems_event_send(taskId, RTEMS_EVENT_7) ;

    flag = !flag ;

    rtems_timer_reset(timer_id) ;
}

rtems_task task1( rtems_task_argument ignored )
{
    rtems_event_set event_out ;

    while (1)
    {
        event_out = 0 ;

        /*wait on event 7 and 8*/
        rtems_event_receive (RTEMS_EVENT_7 | RTEMS_EVENT_8,
                             RTEMS_WAIT | RTEMS_EVENT_ANY,
                             RTEMS_NO_TIMEOUT, &event_out) ;

        /*detect wich event have been received*/
        if ((event_out & RTEMS_EVENT_7) == RTEMS_EVENT_7)
            printf("Event received : RTEMS_EVENT_7\n") ;
        else if ((event_out & RTEMS_EVENT_8) == RTEMS_EVENT_8)
            printf("Event received : RTEMS_EVENT_8\n") ;

        fflush(stdout) ;
    }
}
```

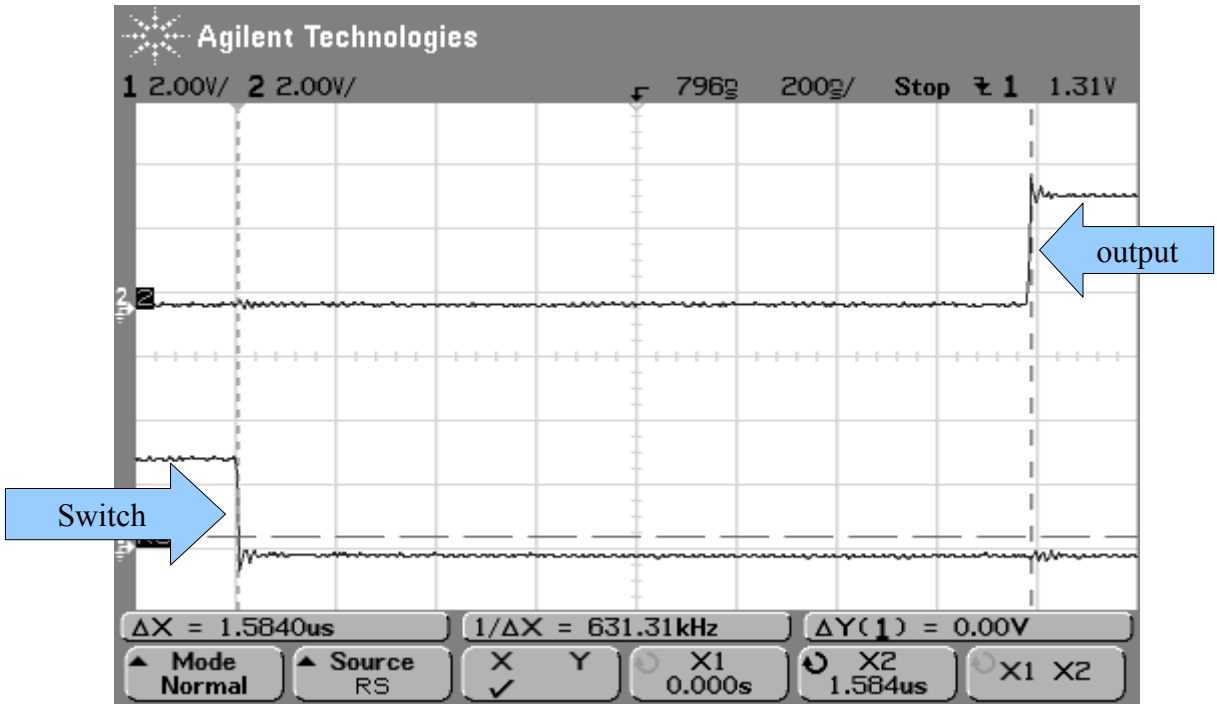
```c
/*task to Init the system*/
rtems_task Init(rtems_task_argument ignored)
{
    rtems_status_code status;
    rtems_id timerId ;
    rtems_id idt1 ;

    /*create and start the task*/
    status = rtems_task_create(
            rtems_build_name( 'T', 'A', '1', ' ' ), 1,
            RTEMS_MINIMUM_STACK_SIZE, RTEMS_DEFAULT_MODES,
            RTEMS_DEFAULT_ATTRIBUTES | RTEMS_FLOATING_POINT, &idt1);
    assert( !status );

    status = rtems_task_start( idt1, task1, 0);
    assert( !status );

    /*create and start the timer*/
    status = rtems_timer_create (rtems_build_name( 'T', 'A', '1', ' ' ), &timerId);
    assert( !status ) ;

    /*the task's id is given to the timer's function*/
    status = rtems_timer_fire_after(timerId, 100, timer1Hz, &idt1);
    assert( !status );

    flag = TRUE ;

    status = rtems_task_delete( RTEMS_SELF ) ;
    exit( 0 );
}


/* configuration information */
#define CONFIGURE_MICROSECONDS_PER_TICK RTEMS_MILLISECONDS_TO_MICROSECONDS(10)
#define CONFIGURE_TEST_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_TEST_NEEDS_CLOCK_DRIVER
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_MAXIMUM_TIMERS 1
#define CONFIGURE_MAXIMUM_TASKS 3

#define CONFIGURE_INIT

#include <rtems/confdefs.h>

/* end of file */
```

```
/*** Annexe 2.9 ***/
/*****************************************************************************/
/* FILENAME       :     osEntry.c                                            */
/* AUTHOR         :     Rochat Joel                                          */
/*---------------------------------------------------------------------------*/
/* FUNCTION       :     Test of interruptions                                */
/*---------------------------------------------------------------------------*/
/* Description    :     An interruption is connected to the IRQ5 entry of    */
/*                      the cpu.This irq occures when the s4 switch of the   */
/*                      ARMEBS3 is pressed. To controle the latency time we  */
/*                      set a pio output and we watch it with an oscillator. */
/*                      The BSP functions are used here.                     */
#undef  NDEBUG

#include <irq.h>
#include <bsp.h>
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <at91rm9200_gpio.h>

#define AT91C_PIO_PD1 ((unsigned int) 1 <<  1)

int gTick ;
int sec ;

rtems_task task1( rtems_task_argument ignored )
{
      while (1)
      {
            if (gTick)
            {
                  printf("ISR came\n") ;
                  gTick = FALSE ;
            }
            else
            {
                  printf("t1") ;
            }

            fflush(stdout) ;
            rtems_task_wake_after(10) ;
      }
}

/*function called when irq occures*/
static void Irq5_handler (uint32_t vector)
{
      /*set pio output*/
      PIOD_REG(PIO_SODR) = AT91C_PIO_PD1 ;
      PIOD_REG(PIO_CODR) = AT91C_PIO_PD1 ;

      gTick = TRUE ;

      /*clear interrupt*/
      AIC_CTL_REG(AIC_ICCR) = 1 << AT91RM9200_INT_IRQ5 ;
      AIC_CTL_REG(AIC_EOICR) = 1 << AT91RM9200_INT_IRQ5 ;
}
```

```
static void Irq5_isr_on (rtems_irq_connect_data *irq_isr_data)
{
      printf("IRQ is on\n") ;
      fflush(stdout) ;
}

static void Irq5_isr_off (rtems_irq_connect_data *irq_isr_data)
{}

static int Irq5_isr_is_on (rtems_irq_connect_data *irq_isr_data)
{
      return 1 ;
}

/*function used to install the irq*/
int connect_irq (rtems_irq_connect_data *irq_isr_data)
{
      /*priority and level trigger muss be configured manually for arm cpus*/
      AIC_SMR_REG(AIC_SMR_IRQ5) = AIC_SMR_PRIOR(irq_isr_data->irqLevel)
                                                | irq_isr_data->irqTrigger ;

      /*IRQ installation*/
    if (!BSP_install_rtems_irq_handler(irq_isr_data))
      return -1 ;

    return 0 ;
}

rtems_task Init (rtems_task_argument ignored)
{
      rtems_status_code status;
      rtems_id          id ;

      /*create and start the task*/
      status = rtems_task_create(
            rtems_build_name( 'T', 'A', '1', ' ' ), 1,
            RTEMS_MINIMUM_STACK_SIZE, RTEMS_DEFAULT_MODES,
            RTEMS_DEFAULT_ATTRIBUTES | RTEMS_FLOATING_POINT, &id);
      assert( !status );

      status = rtems_task_start( id, task1, 0);
      assert( !status );

      /*configure I/O*/
      PIOD_REG(PIO_PER) = AT91C_PIO_PD1 ;
      PIOD_REG(PIO_OER) = AT91C_PIO_PD1 ;
      PIOD_REG(PIO_CODR) = AT91C_PIO_PD1 ;

      /*build irq data*/
      rtems_irq_connect_data irq_isr_data =
      {
           AT91RM9200_INT_IRQ5, // IRQ number
           Irq5_handler,      //IRQ Handler
           Irq5_isr_on,
           Irq5_isr_off,      // Other functions
           Irq5_isr_is_on,
           2,                        //priority level
           AIC_SMR_SRC_EDGE_LOW //trigger methode
      };
```

```
        /*IRQ connection*/
        if (connect_irq (&irq_isr_data) == -1)
              printf("Error connecting IRQ\n") ;
        else
              printf("Connection okay\n") ;

        fflush(stdout) ;

        gTick = FALSE ;
        status = rtems_task_delete( RTEMS_SELF );
        exit( 0 );
}


/* configuration information */
#define CONFIGURE_MICROSECONDS_PER_TICK 1000
#define CONFIGURE_TEST_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_TEST_NEEDS_CLOCK_DRIVER
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_MAXIMUM_TIMERS 1
#define CONFIGURE_MAXIMUM_TASKS 3

#define CONFIGURE_INIT

#include <rtems/confdefs.h>

/* end of file */
```

The first entry is connected to the switch. The second entry is connected to the output pin selected. The latency time is about 1.6us.

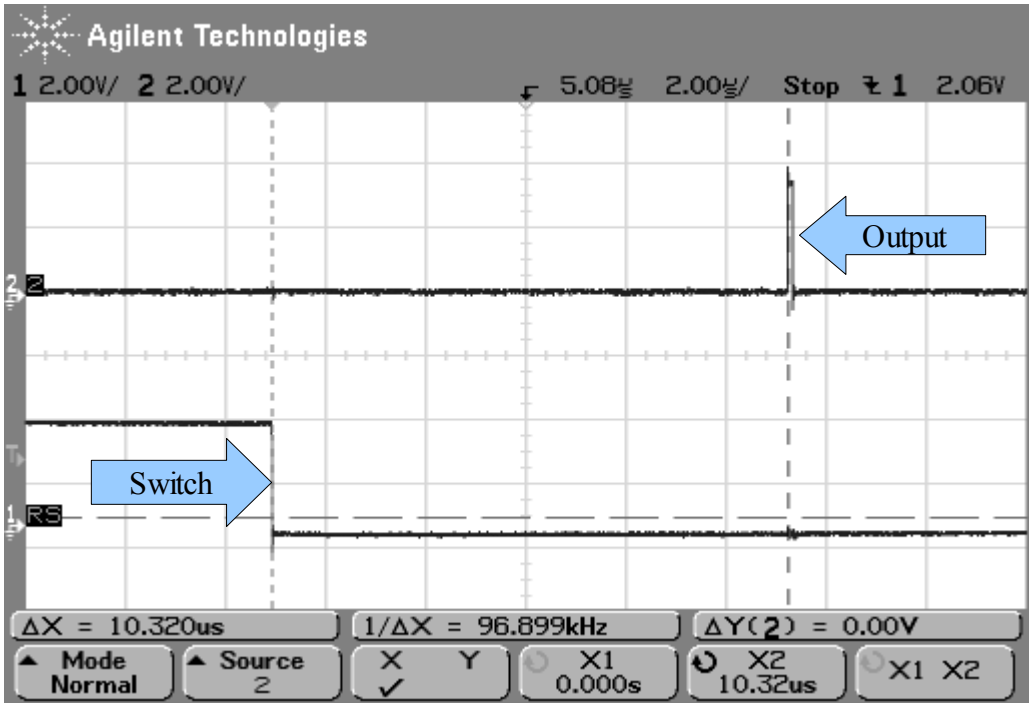


```
/***  Annexe 2.10 ***/
/************************************************************************** */
/* FILENAME       :     osEntry.c                                          */
/* AUTHOR         :     Rochat Joel                                        */
/*-------------------------------------------------------------------------*/
/* FUNCTION       :     Test of interruptions                              */
/*-------------------------------------------------------------------------*/
/* Description    :     Another interruption exemple. This time the isr    */
/*                      handler send an event and the task1 do the job.    */
/*                      We will see the time used to send and receive the  */
/*                      event.                                             */
/*************************************************************************/
#undef  NDEBUG

#include <irq.h>
#include <bsp.h>
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <at91rm9200_gpio.h>

#define AT91C_PIO_PD1 ((unsigned int) 1 <<  1)

rtems_id idTask1 ;

rtems_task task1( rtems_task_argument ignored )
{
      rtems_event_set event_out ;

      while (1)
      {
            event_out = 0 ;

            /*wait on event 7*/
            rtems_event_receive (RTEMS_EVENT_7, RTEMS_WAIT,
                                       RTEMS_NO_TIMEOUT, &event_out) ;

            /*set pio output*/
            PIOD_REG(PIO_SODR) = AT91C_PIO_PD1 ;
            PIOD_REG(PIO_CODR) = AT91C_PIO_PD1 ;
      }
}

/*function called when irq occures*/
static void Irq5_handler (uint32_t vector)
{
      rtems_event_send(idTask1, RTEMS_EVENT_7) ;

      /*clear interrupt*/
      AIC_CTL_REG(AIC_ICCR) = 1 << AT91RM9200_INT_IRQ5 ;
      AIC_CTL_REG(AIC_EOICR) = 1 << AT91RM9200_INT_IRQ5 ;
}

static void Irq5_isr_on (rtems_irq_connect_data *irq_isr_data)
{
      printf("IRQ is on\n") ;
      fflush(stdout) ;
}

static void Irq5_isr_off (rtems_irq_connect_data *irq_isr_data)
{}
```

```
static int Irq5_isr_is_on (rtems_irq_connect_data *irq_isr_data)
{
      return 1 ;
}

/*function used to install the irq*/
int connect_irq (rtems_irq_connect_data *irq_isr_data)
{
      /*priority and level trigger muss be configured manually for arm cpus*/
      AIC_SMR_REG(AIC_SMR_IRQ5) = AIC_SMR_PRIOR(irq_isr_data->irqLevel)
                                                | irq_isr_data->irqTrigger ;

      /*IRQ installation*/
    if (!BSP_install_rtems_irq_handler(irq_isr_data))
      return -1 ;

    return 0 ;
}

rtems_task Init (rtems_task_argument ignored)
{
      rtems_status_code status;
      rtems_id id ;

      /*create and start the task*/
      status = rtems_task_create(
            rtems_build_name( 'T', 'A', '1', ' ' ), 1,
            RTEMS_MINIMUM_STACK_SIZE, RTEMS_DEFAULT_MODES,
            RTEMS_DEFAULT_ATTRIBUTES | RTEMS_FLOATING_POINT, &id);
      assert( !status );

      status = rtems_task_start( id, task1, 0);
      assert( !status );

      /*configure I/O*/
      PIOD_REG(PIO_PER) = AT91C_PIO_PD1 ;
      PIOD_REG(PIO_OER) = AT91C_PIO_PD1 ;
      PIOD_REG(PIO_CODR) = AT91C_PIO_PD1 ;

      /*build irq data*/
      rtems_irq_connect_data irq_isr_data =
      {
           AT91RM9200_INT_IRQ5, // IRQ number
           Irq5_handler,      //IRQ Handler
           Irq5_isr_on,
           Irq5_isr_off,      // Other functions
           Irq5_isr_is_on,
           2,                         //priority level
           AIC_SMR_SRC_EDGE_LOW //trigger methode
      };

      /*IRQ connection*/
      if (connect_irq (&irq_isr_data) == -1)
            printf("Error connecting IRQ\n") ;
      else
            printf("Connection okay\n") ;

      fflush(stdout) ;

      idTask1 = id ;

      status = rtems_task_delete( RTEMS_SELF );
      exit( 0 );
```

```
}


/* configuration information */
#define CONFIGURE_MICROSECONDS_PER_TICK 1000
#define CONFIGURE_TEST_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_TEST_NEEDS_CLOCK_DRIVER
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_MAXIMUM_TIMERS 1
#define CONFIGURE_MAXIMUM_TASKS 3

#define CONFIGURE_INIT

#include <rtems/confdefs.h>
```

The first entry is connected to the switch. The second entry is connected to the output pin selected. But this time, a task waits on an event and set the output when the event occurres. The event is send by the timer. We can see that the latency time is about 10us.



```c
/*** Annexe 2.11 ***/
/*****************************************************************************/
/* FILENAME     :   osEntry.c                                                */
/* AUTHOR       :   Rochat Joel                                              */
/*---------------------------------------------------------------------------*/
/* FUNCTION     :   Test of system clock management                          */
/*---------------------------------------------------------------------------*/
/* Description  :   In the init task the system clock is set. The task1 wake up */
/*                  the first day of the year                                */
/*****************************************************************************/

#undef  NDEBUG

#include <bsp.h>
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

/*print formatted date*/
void printDate (rtems_time_of_day *time) ;

rtems_task task1 (rtems_task_argument ignored)
{
    rtems_time_of_day time ;
    rtems_time_of_day wakeDate ;

    /*build first day of the year*/
    wakeDate.year   = 2008 ;
    wakeDate.month  = 01 ;
    wakeDate.day    = 01 ;
    wakeDate.hour   = 00 ;
    wakeDate.minute = 00 ;
    wakeDate.second = 00 ;
    wakeDate.ticks  = 0 ;

    while(1)
    {
        /*sleep until new year*/
        rtems_task_wake_when(&wakeDate) ;

        printf ("happy new year ! ") ;
        rtems_clock_get ( RTEMS_CLOCK_GET_TOD, &time );
        printDate (&time) ;

        /*End of task*/
        printf ("*** END OF SYSTEM CLOCK TEST ***") ;
        fflush (stdout) ;
        rtems_task_suspend (RTEMS_SELF) ;
    }
}

rtems_task Init (rtems_task_argument ignored)
{
    rtems_status_code status ;
    rtems_id          id ;
    rtems_time_of_day time ;

    /*build the date*/
    time.year   = 2007;
    time.month  = 12;
    time.day    = 31;
    time.hour   = 23;
    time.minute = 59;
    time.second = 55;
    time.ticks  = 0;
```

```c
    /*set date*/
    status = rtems_clock_set(&time) ;

    /*get actual time and print it*/
    rtems_clock_get ( RTEMS_CLOCK_GET_TOD, &time );
    printf ("System clock set : ") ;
    printDate (&time) ;


    /*Create and start the first task*/
    status = rtems_task_create(
        rtems_build_name( 'T', 'A', '1', ' ' ), 1,
        RTEMS_MINIMUM_STACK_SIZE, RTEMS_DEFAULT_MODES,
        RTEMS_DEFAULT_ATTRIBUTES | RTEMS_FLOATING_POINT, &id);
    assert( !status );

    status = rtems_task_start( id, task1, 0 );
    assert( !status );

    status = rtems_task_delete( RTEMS_SELF );

    exit( 0 );
}

void printDate (rtems_time_of_day *time)
{
    printf ("%d-%d-%d  %d:%d:%d\n",
            time->day, time->month, time->year,
            time->hour, time->minute, time->second) ;
    fflush (stdout) ;
}

/* configuration information */
#define CONFIGURE_MICROSECONDS_PER_TICK RTEMS_MILLISECONDS_TO_MICROSECONDS(10)
#define CONFIGURE_TEST_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_TEST_NEEDS_CLOCK_DRIVER
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_MAXIMUM_TASKS 3

#define CONFIGURE_INIT

#include <rtems/confdefs.h>
```

```c
/*** Annexe 2.12 ***/
/*****************************************************************************/
/* FILENAME     :   osEntry.c                                                */
/* AUTHOR       :   Rochat Joel                                              */
/*---------------------------------------------------------------------------*/
/* FUNCTION     :   Memory Partition test                                    */
/*---------------------------------------------------------------------------*/
/* Description  :   One partition allocate PARTITON_SIZE bytes in the init task.*/
/*                  This partition starts at address FREE_MEM in the ram and */
/*                  is able to give BUFFER_SIZE bytes each time the          */
/*                  rtems_partition_get_buffer is called                     */
/*****************************************************************************/

#undef  NDEBUG

#include <bsp.h>
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

/*Free memory address start*/
#define FREE_MEM 0x20300000

/*Size of the partition*/
#define PARTITON_SIZE 1024

/*Size of the returned buffer
 * when rtems_partition_get_buffer is called*/
#define BUFFER_SIZE 16

rtems_task task1 (rtems_id partId)
{
    rtems_status_code status ;
    char *buf ;
    int i = 0 ;

    while (1)
    {
        //get 16 bytes
        status = rtems_partition_get_buffer (partId, (void *)&buf) ;

        if (status == RTEMS_SUCCESSFUL)
        {
            printf ("address : %x allocated\n", buf) ;
            fflush (stdout) ;
            i++ ;
        }
        else
        {
            /*Ther partition is full*/
            printf ("Partition is full : %d bytes allocated",
                    i*BUFFER_SIZE) ;
            fflush (stdout) ;
            rtems_task_suspend (RTEMS_SELF) ;
        }
    }
}
```

```
rtems_task Init (rtems_task_argument ignored)
{
    rtems_status_code status ;
    rtems_id id ;
    rtems_id partId ;

    /*create partition*/
    status = rtems_partition_create(
        rtems_build_name( 'P', 'A', 'R', ' ' ),
        (void *)FREE_MEM,   //start of memory
        PARTITON_SIZE,      //1 kb
        BUFFER_SIZE,        //buffer size = 16 bytes
        RTEMS_DEFAULT_ATTRIBUTES,
        &partId) ;
    assert( !status );

    printf ("The partition starts at %x\n", FREE_MEM) ;
    printf ("Here size is %d bytes\n", PARTITON_SIZE) ;
    printf ("The buffer size is %d bytes\n", BUFFER_SIZE) ;
    fflush (stdout) ;

    /*Create and start the first task*/
    status = rtems_task_create(
        rtems_build_name( 'T', 'A', '1', ' ' ), 1,
        RTEMS_MINIMUM_STACK_SIZE, RTEMS_DEFAULT_MODES,
        RTEMS_DEFAULT_ATTRIBUTES | RTEMS_FLOATING_POINT, &id);
    assert( !status );

    status = rtems_task_start( id, task1, partId);
    assert( !status );

    status = rtems_task_delete( RTEMS_SELF );

    exit( 0 );
}

/* configuration information */
#define CONFIGURE_MAXIMUM_PARTITIONS 2
#define CONFIGURE_TEST_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_TEST_NEEDS_CLOCK_DRIVER
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_MAXIMUM_TASKS 2

#define CONFIGURE_INIT

#include <rtems/confdefs.h>
```

```
/*** Annexe 2.13 ***/
/*****************************************************************************/
/* FILENAME     :   init.c                                                   */
/* AUTHOR       :   Rochat Joel                                              */
/*---------------------------------------------------------------------------*/
/* FUNCTION     :   Test of network and ntp                                  */
/*---------------------------------------------------------------------------*/
/* Description  :   The init Task initialize the network
/*****************************************************************************/

#include <bsp.h>

#define CONFIGURE_TEST_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_TEST_NEEDS_CLOCK_DRIVER
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS 20
#define CONFIGURE_USE_IMFS_AS_BASE_FILESYSTEM

#define CONFIGURE_EXECUTIVE_RAM_SIZE    (512*1024)
#define CONFIGURE_MAXIMUM_SEMAPHORES    20
#define CONFIGURE_MAXIMUM_TASKS     20

#define CONFIGURE_MICROSECONDS_PER_TICK 10000

#define CONFIGURE_INIT_TASK_STACK_SIZE  (10*1024)
#define CONFIGURE_INIT_TASK_PRIORITY    4
#define CONFIGURE_INIT_TASK_INITIAL_MODES (RTEMS_PREEMPT | \
                                           RTEMS_NO_TIMESLICE | \
                                           RTEMS_NO_ASR | \
                                           RTEMS_INTERRUPT_LEVEL(0))

#define CONFIGURE_INIT
rtems_task Init (rtems_task_argument argument);

#include <rtems/confdefs.h>
#include <at91rm9200.h>
#include <at91rm9200_emac.h>


#include <stdio.h>
#include <rtems/rtems_bsdnet.h>
#include <rtems/error.h>
#include "networkconfig.h"

/* our MAC_ADDRESS is 00:00:17:AE:30:00*/
#define MAC_HIGH    ((0xAE << 24) | (0x17 << 16) | (0x00 << 8) | 00)
#define MAC_LOW     ((0x23 << 8) | 0x30)

/*set the Ethernet registers*/
void setMACaddress() ;

/*print formatted date*/
void printDate (rtems_time_of_day *time) ;

/*
 * RTEMS Startup Task
 */
rtems_task Init (rtems_task_argument ignored)
{
    int i ;
    rtems_status_code sc ;
    rtems_time_of_day now ;
    rtems_interval ticksPerSecond ;
    int rtems_bsdnet_synchronize_ntp (int interval, rtems_task_priority priority) ;
```

```
    printf ("****************** NTP TEST **************\n");

    /*set mac adress*/
    setMACaddress() ;

    rtems_bsdnet_initialize_network () ;
    /*get time*/

    printf ("Get time from ntp server...\n");

    rtems_bsdnet_synchronize_ntp (0, 0);

    sc = rtems_clock_get (RTEMS_CLOCK_GET_TOD, &now);

    if (sc != RTEMS_SUCCESSFUL)
        printf ("Failed to get time of day: %s\n", rtems_status_text (sc));

    /*print time received*/
    printDate (&now) ;

    exit (0);
}

void setMACaddress ()
{
    (EMAC_REG(EMAC_SA1L) ) = MAC_HIGH ;
    (EMAC_REG(EMAC_SA1H) ) = MAC_LOW ;
}

void printDate (rtems_time_of_day *time)
{
    printf ("%d-%d-%d  %d:%d:%d\n",
            time->day, time->month, time->year,
            time->hour, time->minute, time->second) ;
    fflush (stdout) ;
}
```

```
/*** Annexe 2.14 ***/
/*****************************************************************************/
/* FILENAME     :   networkconfig.h                                          */
/* AUTHOR       :   Rochat Joël                                              */
/*---------------------------------------------------------------------------*/
/* Description  :   Configuration of the network                             */
/*****************************************************************************/

#ifndef _RTEMS_NETWORKCONFIG_H_
#define _RTEMS_NETWORKCONFIG_H_

/*
 *  The following will normally be set by the BSP if it supports
 *  a single network device driver.  In the event, it supports
 *  multiple network device drivers, then the user's default
 *  network device driver will have to be selected by a BSP
 *  specific mechanism.
 */

#ifndef RTEMS_BSP_NETWORK_DRIVER_NAME
#warning "RTEMS_BSP_NETWORK_DRIVER_NAME is not defined"
#define RTEMS_BSP_NETWORK_DRIVER_NAME "no_network1"
#endif

#ifndef RTEMS_BSP_NETWORK_DRIVER_ATTACH
#warning "RTEMS_BSP_NETWORK_DRIVER_ATTACH is not defined"
#define RTEMS_BSP_NETWORK_DRIVER_ATTACH 0
#endif

/* #define RTEMS_USE_BOOTP */

#include <bsp.h>

/*
 * Define RTEMS_SET_ETHERNET_ADDRESS if you want to specify the
 * Ethernet address here.  If RTEMS_SET_ETHERNET_ADDRESS is not
 * defined the driver will choose an address.
 */
#define RTEMS_SET_ETHERNET_ADDRESS
#if (defined (RTEMS_SET_ETHERNET_ADDRESS))
static char ethernet_address[6] = { 0x00, 0x00, 0x17, 0xae, 0x30, 0x00};
#endif

#ifdef RTEMS_USE_LOOPBACK
/*
 * Loopback interface
 */
extern void rtems_bsdnet_loopattach();
static struct rtems_bsdnet_ifconfig loopback_config = {
    "lo0",              /* name */
    rtems_bsdnet_loopattach,    /* attach function */

    NULL,               /* link to next interface */

    "127.0.0.1",            /* IP address */
    "255.0.0.0",            /* IP net mask */
};
#endif

/*
 * Default network interface
 */
static struct rtems_bsdnet_ifconfig netdriver_config = {
    RTEMS_BSP_NETWORK_DRIVER_NAME,      /* name */
    RTEMS_BSP_NETWORK_DRIVER_ATTACH,    /* attach function */
```

```
#ifdef RTEMS_USE_LOOPBACK
    &loopback_config,       /* link to next interface */
#else
    NULL,               /* No more interfaces */
#endif

#if (defined (RTEMS_USE_BOOTP))
    NULL,               /* BOOTP supplies IP address */
    NULL,               /* BOOTP supplies IP net mask */
#else
    "153.109.5.177",        /* IP address */
    "255.255.255.0",        /* IP net mask */
#endif /* !RTEMS_USE_BOOTP */

#if (defined (RTEMS_SET_ETHERNET_ADDRESS))
    ethernet_address,               /* Ethernet hardware address */
#else
    NULL,                           /* Driver supplies hardware address */
#endif
    0               /* Use default driver parameters */
};

/*
 * Network configuration
 */
struct rtems_bsdnet_config rtems_bsdnet_config = {
    &netdriver_config,

#if (defined (RTEMS_USE_BOOTP))
    rtems_bsdnet_do_bootp,
#else
    NULL,
#endif

    5,          /* Default network task priority */
    128*1024,           /* Default mbuf capacity */
    256*1024,           /* Default mbuf cluster capacity */

#if (!defined (RTEMS_USE_BOOTP))
    "rtems_host",       /* Host name */
    "nodomain.com",     /* Domain name */
    "153.109.5.1",  /* Gateway */
    "153.109.5.1",  /* Log host */
    {"212.101.0.10" },  /* Name server(s) */
    {"195.216.64.208"}, /* NTP server(s) */
#endif /* !RTEMS_USE_BOOTP */

};

/*
 * For TFTP test application
 */
#if (defined (RTEMS_USE_BOOTP))
#define RTEMS_TFTP_TEST_HOST_NAME "BOOTP_HOST"
#define RTEMS_TFTP_TEST_FILE_NAME "BOOTP_FILE"
#else
#define RTEMS_TFTP_TEST_HOST_NAME "255.255.255.255"
#define RTEMS_TFTP_TEST_FILE_NAME "996D05D1.img"
#endif

#endif /* _RTEMS_NETWORKCONFIG_H_ */
```

Annexe 3

Partie Terminal

```java
package hevs.terminal.gui;

/*** Annexe 3.1 ***/
/*****************************************************************************/
/* FILENAME     :   TerminalView.java                                        */
/* MODIFIED BY  :   Joël Rochat                                              */
/*---------------------------------------------------------------------------*/
/* FUNCTION     :   This Class is used to Manage a Terminal into an Eclipse  */
/*                  Plugin                                                   */
/*****************************************************************************/
import java.io.IOException;

public class TerminalView extends ViewPart
{
    private static boolean isFirstOpen = true ;
    private  TextConsoleViewer viewConsol = null ;
    private  IOConsole console = null ;
    private Action actionClear ;
    private CommPort commPort = null ;
    private SerialReader reader = null ;
    private SerialWriter writer = null ;

    public TerminalView()
    {
        if (isFirstOpen)
        {   //we had the thread to the Runtime
            Runtime.getRuntime().addShutdownHook(new KillTerminal()) ;
            isFirstOpen = false ;
        }
    }

    //this class kill terminal thread
    public class KillTerminal extends Thread
    {
        public void run()
        {
            if (reader !=  null)
                reader.halt() ;

            if (writer !=  null)
                writer.halt() ;

            if (commPort != null)
                commPort.close() ;

            System.out.println("kill threads") ;
        }
    }

    void connect ( String portName ) throws Exception
    {
        CommPortIdentifier portIdentifier =
            CommPortIdentifier.getPortIdentifier(portName);

        if ( portIdentifier.isCurrentlyOwned() )
        {
            System.out.println("Error: Port is currently in use");
        }
        else
        {
            commPort = portIdentifier.open(this.getClass().getName(),2000);

            if ( commPort instanceof SerialPort )
            {
                //new connection
                SerialPort serialPort = (SerialPort) commPort;
```

```
            //give communication infos
            serialPort.setSerialPortParams(115200,SerialPort.DATABITS_8,
                    SerialPort.STOPBITS_1,SerialPort.PARITY_NONE);
            //get streams
            InputStream in = serialPort.getInputStream();
            OutputStream out = serialPort.getOutputStream();

            //new reader thread
            reader = new SerialReader(in) ;
            reader.start() ;
            //new writer thread
            writer = new SerialWriter(out) ;
            writer.start() ;
        }
        else
        {
            System.out.println("Error: ");
        }
    }
}

public void createPartControl(Composite parent)
{
    //build console
    console = new IOConsole("name", null, null);
    viewConsol = new TextConsoleViewer(parent, console) ;
    createActionClear() ;

    try {
        connect ("COM1") ;
    } catch (Exception e) {

        e.printStackTrace();
    }
}

//kill thread and free com1
public void kill()
{
    if (reader !=  null)
        reader.halt() ;

    while (reader.isAlive()){}

    if (writer !=  null)
        writer.halt() ;

    while (writer.isAlive()){}

    if (commPort != null)
        commPort.close() ;

    writer = null ;
    reader = null ;
    commPort = null ;
}

 public void dispose()
 {
     kill() ;
 }

public void createActionClear()
{
    actionClear = new Action("Clear console") {
        public void run() {
```

```
                console.clearConsole() ;
            }
        };

        MenuManager menuMgr = new MenuManager("#PopupMenu");
        menuMgr.setRemoveAllWhenShown(true);
        menuMgr.addMenuListener(new IMenuListener() {
            public void menuAboutToShow(IMenuManager manager) {
                manager.add(actionClear);
            }
        });
        Menu menu = menuMgr.createContextMenu(viewConsol.getControl());
        viewConsol.getControl().setMenu(menu);
        getSite().registerContextMenu(menuMgr, viewConsol);
    }

    public void setFocus() {

    }

    public class SerialReader extends Thread
    {
        private boolean stopThread = false ;
        private InputStream in ;
        private PrintStream p  ;

        public SerialReader ( InputStream in)
        {
            this.in =in ;
            p = new PrintStream(console.newOutputStream()) ;
        }

        public void run ()
        {
            boolean fin = false;
            byte[] buffer = new byte[1024];
            int len = -1;
            try
            {
                while (!fin && (len = this.in.read(buffer)) > -1)
                {
                    if (len > 0)
                    {
                        p.print(new String(buffer,0,len));
                        System.out.println(new String(buffer,0,len)) ;
                        Display.getDefault().asyncExec(new Runnable()
                        {
                            public void run()
                            {
                                StyledText textWidget = viewConsol.getTextWidget() ;

                                textWidget.setTopIndex(textWidget.getLineCount()) ;
                                textWidget.setCaretOffset(textWidget.getLineCount()) ;
                            }
                          });
                    }
                    synchronized(this)
                    {
                        fin = this.stopThread ;
                    }
                }

                System.out.println("fin reader") ;
            }
            catch ( IOException e )
            {
```

```
                e.printStackTrace();
                System.out.println("error close") ;
            }
        }

        public synchronized void halt()
        {
            //close the stream from reader
            try
            {
                this.stopThread = true ;
                in.close();
                p.close() ;
            }
            catch (Exception e){System.out.println("Error") ;}
        }
    }

    public class SerialWriter extends Thread
    {
        private OutputStream out;
        private InputStream in ;
        private boolean stopThread = false ;

        public SerialWriter ( OutputStream out)
        {
            this.out = out;
            this.in = console.getInputStream() ;
        }

        public void run ()
        {
            boolean fin = false ;
            try
            {
                int c = 0 ;
                while ( !fin && ( c = in.read()) > -1 )
                {
                    this.out.write(c);

                    synchronized(this)
                    {
                        fin = this.stopThread ;
                    }
                }
                System.out.println("fin writer") ;
            }
            catch ( IOException e )
            {
                e.printStackTrace();
            }
        }

        public synchronized void halt()
        {
            //close the stream from writer
            try
            {
                this.stopThread = true ;
                in.close();
                out.close() ;
            }
            catch (Exception e){System.out.println("Error") ;}
        }
    }
}
```