

L1.1 - Document Cas d'Utilisations et Exigences applicables à l'Environnement et aux Outils de Développement



Document rédigé par :	Approuvé par :	Approuvé par :	Approuvé par :
Nicolas Raynaud (Sagem Communications) Olivier Gallot (Axupteam)			

Liste des évolutions

Edition	Date	Evolutions

Table des matières

1 - Introduction.....	7
2 - Périmètre du sous-projet 1.....	7
2.1 - Outil de génération de firmware.....	7
2.2 - Outils de debug, test et mesures.....	7
2.3 - IDE basé sur éclipse.....	8
3 - Nomenclature.....	9
3.1 - Versions de développement et version officielle.....	9
3.2 - Les recettes.....	9
3.3 - Le plan de version.....	9
3.4 - Notion de profils.....	10
3.5 - Processus de production d'un Firmware.....	11
3.5.1 - Analyse du plan de version.....	11
3.5.2 - Fabrication des outils de cross-compilation.....	12
3.5.2.1 - Pour chaque module outils.....	12
3.5.3 - Fabrication des firmwares.....	13
3.5.3.1 - Pour chaque module.....	13
3.5.3.2 - Fabrication de filesystem embarqués.....	14
3.5.3.3 - Packaging du firmware.....	14
4 - Énumération des exigences.....	15
4.1 - Exigences communes.....	15
4.2 - Plan de version.....	15
4.3 - Utilisation de l'IDE.....	15
4.3.1 - Ergonomie.....	15
4.4 - Édition.....	15
4.4.1 - Édition du plan de version	15
4.4.2 - Edition des recettes.....	16
4.4.3 - Edition des sources du projet.....	17
4.5 - Gestion de configuration.....	17
4.6 - Génération.....	17
4.7 - Actions de nettoyage.....	17
4.8 - Exploitation du firmware.....	18
4.9 - Debug.....	18
4.10 - Analyse.....	18
4.11 - Mesure.....	18
4.12 - licences	19
5 - Notation des exigences.....	20
6 - Cas d'utilisations de l'environnement et outils de développements.....	23
7 - Un use case typique.....	23
8 - Description des uses cases.....	25
8.1 - Outils d'édition.....	25

8.1.1 - Choix de l'éditeur.....	25
8.1.2 - Édition de sources avec syntaxe colorée.....	25
8.1.3 - Navigation dans le code source.....	25
8.1.4 - Collectes des licences	26
8.2 - Génération.....	27
8.2.1 - Configuration du plan de version.....	27
8.2.1.1 - Modification du plan de version : description de la version logicielle	27
8.2.1.2 - Modification du plan de version : choix de la cible et de son architecture.....	27
8.2.1.3 - Modification du plan de version : choix du type de temps Reel.....	27
8.2.1.4 - Modification du plan de version : gestion des modules.....	28
8.2.2 - Générations.....	28
8.2.2.1 - Génération d'un firmware selon son plan de version.....	28
8.2.2.2 - Re-Génération d'un firmware selon son plan de version.....	29
8.2.2.3 - Génération d'outils de cross compilation.....	29
8.2.2.4 - Compilation d'un module.....	29
8.2.2.5 - Re-compilation d'un module.....	30
8.2.2.6 - Construction d'une recette	30
8.2.2.7 - Parallélisation de la génération.....	31
8.2.2.8 - Profils.....	31
8.2.2.9 - Patch d'un module	34
8.2.3 - Informations sur la génération.....	34
8.2.3.1 - Graphe des dépendances d'une construction au sein d'un module.....	34
8.2.3.2 - Graphe des dépendances d'un module.....	34
8.2.3.3 - Exploiter les erreurs de construction.....	35
8.2.3.4 - Création d'un fichier de log des erreurs de construction.....	35
8.3 - Gestion de Configurations à partir de l'IDE	36
8.3.1 - Gestion du plan de version	36
8.3.2 - Gestion des recettes.....	36
8.3.3 - Gestion des sources.....	37
8.4 - Debug.....	37
8.4.1 - Debug d'applications.....	37
8.4.2 - Debug de kernel.....	38
8.4.2.1 - Debug dans le kernel par KGDB ou kdb.....	38
8.4.2.2 - Debug sur machine émulée Qemu.....	38
8.4.2.3 - Debug du noyau avec Kprobes.....	39
8.4.2.4 - Utilisation de support hardware au travers du JTAG.....	39
8.4.2.5 - Debug du noyau avec un dump lkcd.....	40
8.4.2.6 - Debug avec le simulateur Xenomai.....	40
8.5 - Analyse.....	40
8.5.1 - Analyse générique d'événements.....	40
8.5.2 - Analyse des événements avec LTT.....	41
8.5.3 - Analyse de performance avec Oprofile.....	42
8.5.4 - Analyse d'utilisation de la mémoire avec Valgrind.....	42
8.5.5 - Exploration statique d'un binaire.....	43
8.5.6 - Analyse des interactions d'une application avec le système.....	43
8.5.7 - Analyse des statistiques d'ordonnancement avec schedtop.....	43

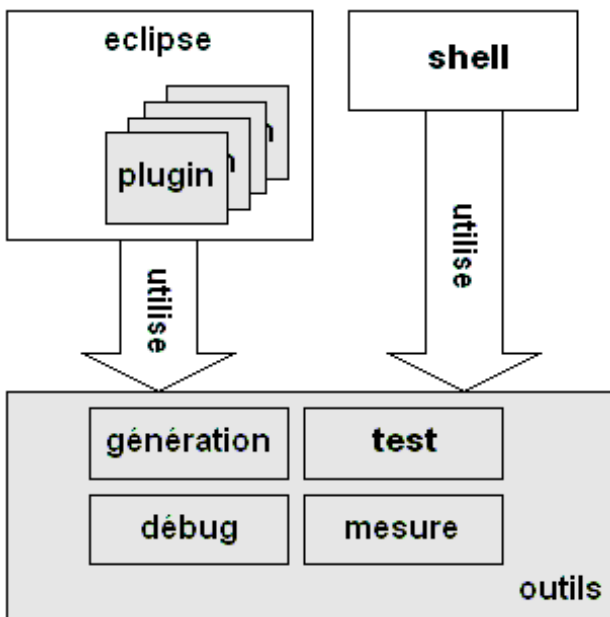
8.5.8 - Analyse des applications multi thread avec PTT.....	44
8.5.9 - Analyse des statistiques d'ordonnancement.....	44
8.6 - Mesure.....	44
8.6.1 - Taux de couverture de code.....	44
8.6.2 - Mesure de latence et gigue.....	45
8.6.3 - Mesure de débit.....	45
8.6.4 - Mesure de la charge du système.....	46
8.6.5 - Mesure de la mémoire.....	46
9 - Notation des cas d'utilisation.....	47

1 - Introduction

Ce document est le livrable associé au sous projet SP1 du projet RTEL4i. Il décrit les divers cas d'utilisation et les exigences que doivent respecter les outils de génération de firmware et les plugin Eclipse associés.

Composé de plusieurs parties, il présente l'esprit général de l'outil tel que nous l'imaginons puis précise le sens de quelques termes. Suit alors une énumération d'exigences accompagnées de leurs notations . C'est alors que sont abordés les cas d'utilisation et une notation de ces cas.

2 - Périmètre du sous-projet 1



Le but du sous projet 1 du projet RTEL4i est la mise en place d'outils permettant à un utilisateur d'aborder facilement la mise en œuvre du temps réel embarqué sous linux.

Ce sous projet met donc en œuvre une collection d'outils permettant la génération du firmware, le debug, le test et la mesure.

Ces outils sont utilisables directement depuis des commandes en ligne. Ils sont aussi utilisables à partir d'une interface graphique, IDE eclipse dont le rôle dans ce projet est de faciliter à l'utilisateur l'accès aux divers technologies offertes.

2.1 - Outil de génération de firmware

Un outil de génération de firmware est un outil permettant de fabriquer un firmware à partir d'une description : «le plan de version ».

L'outil accomplit ensuite les tâches suivantes :

- éventuellement la fabrication et l'installation des outils de cross-compilation
- la fabrication du firmware en faisant de la cross compilation

L'outil de génération peut générer des packages, des filesystem embarquables ou accessibles par le réseau (par nfs), des kernels, des binaires contenant ensemble kernel et filesystem.

L'outil de génération de firmware doit pouvoir construire les firmwares de machines aux architectures variées.

Cet outil doit pouvoir être appelé directement à partir d'un shell.

2.2 - Outils de debug, test et mesures

Ces outils sont la plupart du temps déjà disponibles dans la communauté. Ce sous projet les identifie et les intègre en tant que partie prenante de la distribution.

2.3 - IDE basé sur eclipse

Dans le cadre de ce projet, l'IDE basé sur eclipse est une interface graphique faite pour faciliter l'approche de la génération de firmware aux personnes mal à l'aise avec les commandes en ligne.

Elle doit permettre :

- La création et la modification de plan de version
- Le lancement des compilations de firmwares en s'appuyant sur l'outil de génération de firmware
- Le débog de firmwares sur des plateformes virtualisées ou réelles
- Des analyses et mesures de temps réel

3 - Nomenclature

Un certain nombre de terme sont utilisés dans ce document. Ce chapitre explicite les concepts que recouvrent ces différents termes.

3.1 - Versions de développement et version officielle

Seul un logiciel muni d'une version officielle est susceptible d'être distribué. Les logiciels non munis de version officielle ont une version de développement qui peut être propre au développeur, à un intégrateur ou caractériser une bêta ou une pré-release.

Les versions officielles doivent pouvoir être reconstruites de façon strictement identiques à tout moment. Un plan de version immuable leur est associé. Celui ci décrit strictement, par leurs noms et leurs numéros de version, chacun des outils et composants logiciels nécessaires pour la génération.

3.2 - Les recettes

Certains outils de génération de firmware (OpenWRT et openembedded par exemple) introduisent la notion de recette. Les recettes sont des enregistrements qui regroupent un certain nombre de concepts qui vont permettre de manipuler les différents composants logiciels et outils nécessaires à la construction des firmwares.

Les recettes peuvent être vues comme des objets dont le but est de fournir une couche d'abstraction permettant d'aborder l'hétérogénéité des développements dans le mode de l'opensource.

Pour chaque module d'un projet, il est ainsi possible d'associer une recette qui décrira comment trouver les sources, les patcher, les configurer, les compiler et les installer sur un file system fictif représentant l'arborescence des filesystem de l'embarqué.

- un nom de module
- un numéro de version
- l'URI du module
- La méthode d'accès (svn, cvs, git, tar...)
- une méthode de patch
- une méthode de préparation des sources (qui appelle par exemple un ./configure)
- une méthode de compilation
- une méthode d'installation des résultats sur un filesystem de l'hôte en attendant de fabriquer les filesystems embarqués

3.3 - Le plan de version

Le plan de version est un fichier énumérant les outils et les composants logiciels permettant de construire un firmware.

Il n'y a pas de format privilégié pour décrire un plan de version. Il faut néanmoins que celui ci puisse être lu et modifié par un utilisateur à l'aide d'un simple éditeur

Si l'outil de génération possède le concept de recette (voir le chap «3.2 - Les recettes ») , il prend alors la forme suivante :

- Le nom de la machine cible
- L'architecture du processeur de la machine cible
- Un nom de release pour ce firmware
C'est ici que sont décrits soit le numéro de version officiel, soit le numero de version de développement. Cette information fera partie du firmware, elle permet à l'utilisateur de connaître la version de son logiciel. Le format de ce champs est un texte libre.
- Les outils sont chacun décrits par :
 - un nom de recette
 - un numéro de version de la recette
 - l'URI où peut être trouvé la recette
 - La méthode d'accès (svn, cvs, git, tar...)
- Les composants logiciels sont chacun décrits par :
 - un nom de recette
 - un numéro de version de la recette
 - l'URI où peut être trouvé la recette
 - La méthode d'accès (svn, cvs, git, tar...)

Le plan de version doit pourvoir décrire de façon non ambiguë les outils et modules qui permettent la génération d'un filesystem. Un plan de version totalement décrit doit pouvoir permettre des reconstructions reproductibles de firmware sur des machines de développement non dédiées.

Ainsi, les binaires de 2 firmwares embarqués décrits par un même plan de version et générés sur 2 machines différentes doivent être strictement identiques.

Le plan de version est indépendant de la méthode de construction. Il ne doit pas être confondu avec la notion de projet d'eclipse.

3.4 - Notion de profils

Le plan de version est fait pour être invariable et déterministe. Il apparaît vite que sa manipulation devient fastidieuse lorsqu'il s'agit d'introduire des variations nécessaires lors des phases de développement par exemple pour faire des versions debug, ou pour faire fonctionner cette même version en labo de manière particulière (par exemple un montage NFS sur un filesystem déporté). C'est pourquoi a été évoqué la notion de profil : Une forme du plan de version permettant au développeur de facilement manipuler les variations de son firmware.

L'IDE comme eclipse devient alors l'outil idéal pour exploiter ce concept. Il gère un projet comme un ensemble de profils ayant à la base une même description et profitant des mêmes options que la racine.

Les profils permettent de décrire des options qui leurs sont propres. Ils peuvent décrire le répertoire où se trouvera le code généré, ils peuvent ajouter des options de debug par exemple, invalider l'inclusion d'un module dans la génération

Les profils ont le même niveau fonctionnel que le plan de version.

Projet

```
|-- Profil Debug
|   |-- kernel
|   |-- module1
|   |-- module2
|   |-- module3
|   `-- module4
|-- profil Release    <=> plan d'une version officielle
    |-- kernel
    |-- module
    |-- module2
    |-- module3
    `-- module4 (disabled en version debug)
```

Clonables au sein d'un projet, les profils pourraient offrir un bel espace de liberté au développeur.

3.5 - Processus de production d'un Firmware

La production de firmware pour un embarqué est une opération complexe. Elle prend en compte les contraintes du projet et s'assure que l'environnement de production soit stable.

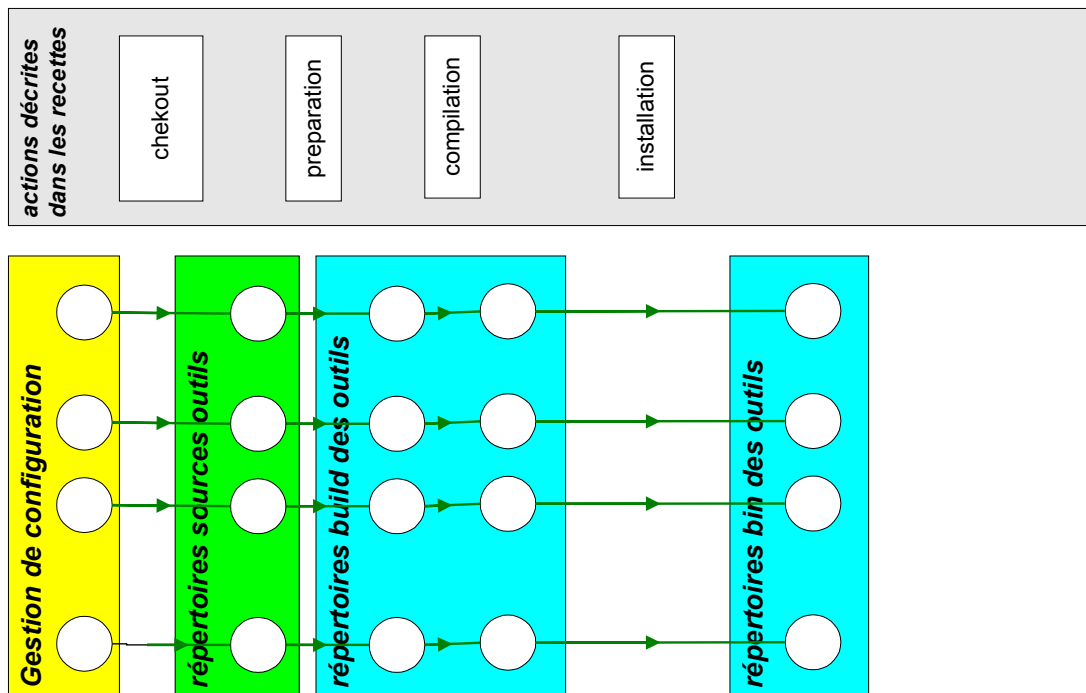
La génération se base sur un outil de génération et s'articule autour du plan de version.

Dans un premier temps est analysé le plan de version, puis sont construits les outils de cross-compilation. A l'aide des outils de cross-compilation sont construits le kernel, les applications, bibliothèques, driver. L'ensemble des binaires construits, sauf le kernel, sont installés dans des images de filesystem. La dernière opération consiste à prendre le kernel et l'ensemble des filesystem pour fabriquer un package destiné à être téléchargé sur la cible.

3.5.1 - Analyse du plan de version

Le plan de version apparaît pour le système de génération de firmware comme un ensemble de consignes à respecter pour générer le firmware.

3.5.2 - Fabrication des outils de cross-compilation



3.5.2.1 - Pour chaque module outils

Les modules outils peuvent être décrits par des sources ou peuvent être déjà construits. Dans ce dernier cas, on ira directement à la phase d'installation

3.5.2.1.1 - Extraction des sources

Cette phase consiste à extraire, pour chaque module, les sources décrites dans le plan de version depuis une gestion de configuration.

3.5.2.1.2 - Application des patch

L'application des patch doit être faite impérativement avant la préparation.

3.5.2.1.3 - Préparation

Cette tâche consiste à collecter les contraintes de l'environnement et à fabriquer des makefile adaptés.

3.5.2.1.4 - Compilation

Tâche de compilation du module lancé à partir des makefiles fabriqués précédemment.

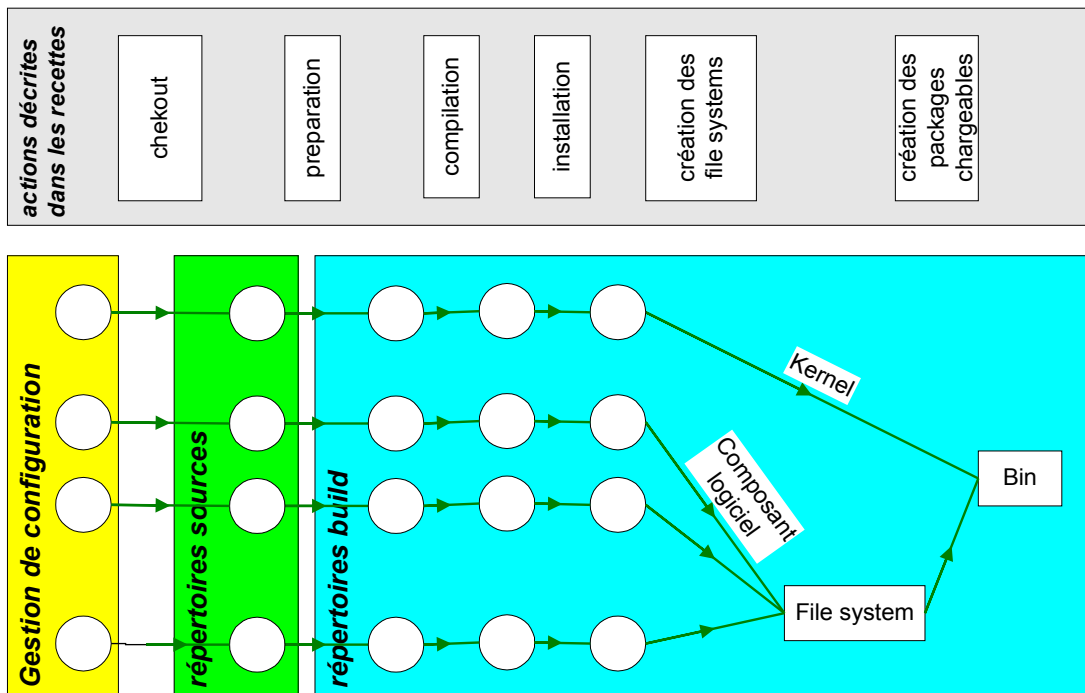
3.5.2.1.5 - Installation

Cette phase consiste à installer dans l'environnement de l'utilisateur les outils nécessaires à la génération des firmwares.

3.5.3 - Fabrication des firmwares

Cette phase ne peut commencer que lorsque les outils de cross-compilation ont été générés et installés. Comme pour la fabrication des outils, la génération de firmware se base sur les informations de plan de version pour :

- les tâches d'extraction de source
- les tâches de préparation et de patch
- les tâches de compilation
- les tâches d'installations
- les tâches de fabrications de filesystem
- les tâches de packaging du firmware



3.5.3.1 - Pour chaque module

Les modules peuvent être décrit par des sources ou peuvent être déjà construits. Dans ce dernier cas, on ira directement à la phase d'installation

3.5.3.1.1 - Extraction des sources

Cette phase consiste à extraire, pour chaque module, les sources décrites dans le plan de version depuis une gestion de configuration, un filesystem ou un espace sur le web.

3.5.3.1.2 - Préparation

Cette tâche consiste à collecter les contraintes de l'environnement et à fabriquer des makefile adaptés.

Important : Lors de cette tâche des informations spécifiques à la version du firmware et contenues dans le plan de version sont susceptibles d'être incluses dans les makefiles.

3.5.3.1.3 - Application des patch

L'application des patch doit être faite impérativement avant la préparation.

3.5.3.1.4 - Compilation

Compilation du module lancé à partir des makefiles fabriqués précédemment.

Important : Lors de cette tâche des informations spécifiques à la version du firmware et contenues dans le plan de version sont susceptibles d'être incluses dans les makefiles.

Elle peut alors être utilisée dans la compilation de certains modules

3.5.3.1.5 - Installation

Cette phase consiste à installer dans une arborescence de répertoire représentant l'ensemble des montages de la cible.

3.5.3.2 - Fabrication de filesystem embarqués

A ce niveau du scénario, le filesystem de la cible est totalement décrit sous la forme d'une arborescence. La fabrication des filesystem embarqués consiste à capter la sous arborescence du filesystem à partir de son point de montage présumé dans l'arborescence générale.

3.5.3.3 - Packaging du firmware

Cette tâche consiste à fabriquer le binaire qui sera chargé tel quel dans la machine. C'est un agrégat contenant le kernel et un ou plusieurs filesystem. Cet agrégat possède une entête compatible avec le bootloader où les informations suivantes sont présentes :

- nom de la release (issus du plan de version)
- un check sum

Il peut arriver que lors du packaging du firmware, plusieurs firmware sensiblement différents soient générés simultanément. Le plus souvent, seul le numero de version et le checksum changent.

4 - Énumération des exigences

Ce chapitre énumère les exigences que doit respecter l'outil de génération de firmware et l'IDE eclipse.

4.1 - Exigences communes

EX_1 : Pouvoir construire facilement un firmware

EX_2 : Pouvoir facilement mettre en œuvre Xenomai ou Premp-RT

EX_3 : Pouvoir facilement mettre en œuvre de façon concurrente plusieurs profils de la même version.

EX_4 : Avoir un outil insensible à l'environnement (l'usage de technique comme chroot est encouragée).

EX_5 : Pouvoir supporter les ARM7

EX_6 : Pouvoir supporter les ARM11

EX_7 : Pouvoir supporter les MIPS32

EX_8 : Pouvoir supporter les PowerPC

EX_9 : Pouvoir supporter les SH4

EX_10 : Pouvoir supporter les SPARC32

4.2 - Plan de version

EX_11 : Le plan de version doit pouvoir être lu et généré par l'IDE.

EX_12 : Le plan de version doit pouvoir être exploité par une commande en ligne depuis un simple shell

EX_13 : Le plan de version doit être lisible par un humain et éditable

EX_14 : Le plan de version doit pouvoir être géré dans une gestion de configuration

EX_15 : Le format du plan de version est indépendant de tout format eclipse

4.3 - Utilisation de l'IDE

4.3.1 - Ergonomie

EX_16 : Hiérarchie des menus doit être intuitive (les commandes groupées par sémantiques voisines)

EX_17 : Fonctions utiles accessibles par des raccourcis clavier

4.4 - Édition

4.4.1 - Édition du plan de version

EX_18 : Permettre la description exhaustive des outils et composants logiciel par le plan de version

- EX_ 19 : Permettre de décrire strictement dans le plan de version les noms et les versions logicielles de chacun des outils et composants logiciels (pour pouvoir faire des version officielles)
- EX_ 20 : Pouvoir mettre un nom de firmware dans le plan de version
- EX_ 20.1 : Pouvoir exploiter le nom du firmware lors de la génération et inclus dans les composants logiciels et les entêtes de packages.
- EX_ 21 : Pouvoir modifier de façon dynamique les divers éléments du plan de version
- EX_ 21.1 : Pouvoir ajouter, retirer, modifier un outil
 - EX_ 21.2 : Pouvoir ajouter, retirer, modifier un des composants logiciel du firmware
 - EX_ 21.3 : Ne pas avoir à tout systématiquement re-générer lorsque le plan de version est modifié
- EX_ 22 : Possibilité de décrire le projet sous la forme d'un overlay par rapport a une base open source
- EX_ 22.1 : La partie open source est alors un clone d'une gestion de configuration de la communauté
 - EX_ 22.2 : La partie spécifique au projet est décrite dans une gestion de configuration propre au projet
 - EX_ 22.3 : Lors de la génération, l'outil de génération de firmware prend le code disponible dans la partie overlay et le complète avec le code de la communauté
- EX_ 23 : Pouvoir mettre en œuvre facilement une distribution temps réel
- EX_ 23.1 : Pouvoir modifier facilement le plan de version pour avoir Xenomai
 - EX_ 23.2 : Pouvoir modifier facilement le plan de version pour choisir les skins de Xenomai
 - EX_ 23.3 : Pouvoir modifier facilement le plan de version pour avoir Preempt-RT
- EX_ 24 : Pouvoir décrire plusieurs profils d'un soft
- EX_ 24.1 : pouvoir invalider certain soft dans un profil
 - EX_ 24.2 : Pouvoir passer des options spécifiques à certains modules dans le cadre d'un profil
 - EX_ 24.3 : Pouvoir passer des options à tout les modules et outils d'un profil
 - EX_ 24.4 : Pouvoir passer des options à tout les modules et outils de tout les profils
 - EX_ 24.5 : Pouvoir exploiter simultanément sur une même machine plusieurs profils sans avoir de conflit.

4.4.2 - Edition des recettes

- EX_ 25 : Pouvoir ajouter ses propres recettes
- EX_ 26 : Pouvoir construire les recettes de façon à pouvoir aborder les techniques de construction les plus communes : make, kbuild, autotools, scons,gmake, cmake

4.4.3 - Edition des sources du projet

EX_27 : Avoir la possibilité d'éditer les sources dans les meilleures conditions possibles

EX_27.1 : colorisation syntaxique en fonction du langage

EX_27.2 : indentation automatique en fonction du langage

EX_27.3 : Naviguer facilement dans le code d'un projet

4.5 - Gestion de configuration

EX_28 : Pouvoir gérer le plan de version sous une gestion de configuration

EX_29 : Pouvoir gérer les recettes dans la gestion de configuration

EX_30 : Pouvoir gérer les outils sous gestion de configuration

EX_31 : Pouvoir gérer les sources des différents composants logiciel sous gestion de configuration

EX_32 : Les sources du projet ne constituent pas une arborescence unique au sein de la gestion de configuration. (chaque outil, chaque composant logiciel, chaque recette, chaque plan de version sont dans des modules de gestion de configuration différents)

4.6 - Génération

EX_33 : L'ensemble du firmware doit pouvoir être généré

EX_33.1 : à partir de l'IDE

EX_33.2 : à partir d'une commande en ligne

EX_34 : Un outil ou un composant logiciel doit pouvoir être compilé de façon unitaire

EX_34.1 : à partir de l'IDE

EX_34.2 : à partir d'une commande en ligne

EX_35 : La génération doit s'arrêter sur une erreur bloquante

EX_36 : l'outil doit aider l'utilisateur à exploiter les erreurs de générations

EX_36.1 : Les outils de colorisation de log sont les bienvenus

EX_37 : l'outil doit pouvoir fabriquer un fichier de log des erreurs

EX_38 : Pouvoir relancer une génération de firmware en ne compilant que ce qui est strictement nécessaire

EX_39 : Pouvoir relancer une génération de module en ne compilant que ce qui est strictement nécessaire

EX_40 : rapidité : il doit être possible d'exploiter les processeurs multi-cores pour accélérer la génération.

4.7 - Actions de nettoyage

EX_41 : Pouvoir effacer simplement tous les fichiers générés lors de la génération

EX_42 : Pouvoir effacer simplement tous les fichiers générés d'un module

EX_43 : Pouvoir effacer simplement tous les fichiers installés dans le pseudo filesystem de sortie

4.8 - Exploitation du firmware

EX_44 : L'IDE doit pouvoir appeler un script ou une application avec les arguments nécessaires pour provoquer le téléchargement du firmware sur la cible.

EX_44.1 : Un bouton facilement accessible ou un champs du menu doit être présent pour permettre cette action

EX_44.2 : un menu de configuration doit être présent pour permettre de décrire le script que l'on souhaite appeler.

4.9 - Debug

EX_45 : Pouvoir déboguer des applications

EX_45.1 : directement sur la cible

EX_45.2 : sur une cible émulée

EX_46 : Pouvoir déboguer au niveau du kernel

EX_46.1 : directement sur la cible

EX_46.2 : sur une cible émulée

EX_46.3 : sur la cible via jtag

EX_47 : Pouvoir mettre en œuvre facilement le simulateur de Xenomai

EX_47.1 : Pouvoir insérer facilement des applications dans le simulateur

4.10 - Analyse

EX_48 : Avoir une vue sur les événements internes de la machine

EX_49 : Avoir une vue sur l'enchaînement des tâches

EX_50 : Avoir une vue sur les appels systèmes

EX_51 : Avoir des chronogrammes avec changement d'échelle

EX_51.1 : Avoir une colorisation des événements

EX_52 : Pouvoir détecter les mauvaises utilisations de mémoire

EX_53 : Pouvoir faire du profiling

4.11 - Mesure

EX_54 : Pouvoir mesurer l'utilisation de la mémoire

EX_55 : Pouvoir faire des mesures de gigue et latence

EX_56 : Pouvoir mesurer la charge du CPU

EX_57 : Pouvoir mesurer le débit au niveau des interfaces

4.12 - licences

EX_58 : Il doit être possible de collecter des informations sur les modules et outils référencés par le plan de version : nom du module, version logicielle utilisée, licence

5 - Notation des exigences

Ce paragraphe décrit l'importance du besoin associé à chacune des exigences

Nom de l'exigence	besoin
EX_ 1 : Pouvoir construire facilement un firmware	fort
EX_ 2 : Pouvoir facilement mettre en œuvre Xenomai ou Premp-RT	fort
EX_ 3 : Pouvoir facilement mettre en œuvre de façon concurrente plusieurs profils de la même version.	fort
EX_ 4 : Avoir un outil insensible à l'environnement (l'usage de technique comme chroot est encouragée).	fort
EX_ 5 : Pouvoir supporter les ARM7	fort
EX_ 6 : Pouvoir supporter les ARM11	fort
EX_ 7 : Pouvoir supporter les MIPS32	fort
EX_ 8 : Pouvoir supporter les PowerPC	fort
EX_ 9 : Pouvoir supporter les SH4	fort
EX_ 10 : Pouvoir supporter les SPARC32	moyen
EX_ 11 : Le plan de version doit pouvoir être lu et généré par l'IDE.	fort
EX_ 12 : Le plan de version doit pouvoir être exploité par une commande en ligne depuis un simple shell	fort
EX_ 13 : Le plan de version doit être lisible par un humain et éditable	fort
EX_ 14 : Le plan de version doit pouvoir être géré dans une gestion de configuration	fort
EX_ 15 : Le format du plan de version est indépendant de tout format eclipse	fort
EX_ 16 : Hiérarchie des menus doit être intuitive (les commandes groupées par sémantiques voisines)	moyen
EX_ 17 : Fonctions utiles accessibles par des raccourcis clavier	faible
EX_ 18 : Permettre la description exhaustive des outils et composants logiciel par le plan de version	fort
EX_ 19 : Permettre de décrire strictement dans le plan de version les noms et les versions logicielles de chacun des outils et composants logiciels (pour pouvoir faire des version officielles)	fort
EX_ 20 : Pouvoir mettre un nom de firmware dans le plan de version	fort
EX_ 21 : Pouvoir modifier de façon dynamique les divers éléments du plan de version	fort
EX_ 22 : Possibilité de décrire le projet sous la forme d'un overlay par rapport a une base open source	moyen

EX_23 : Pouvoir mettre en œuvre facilement une distribution temps réel	fort
EX_24 : Pouvoir décrire plusieurs profils d'un soft	fort
EX_25 : Pouvoir ajouter ses propres recettes	fort
EX_26 : Pouvoir construire les recettes de façon à pouvoir aborder les techniques de construction les plus communes : make, kbuild, autotools, scons, gmake, cmake	fort
EX_27 : Avoir la possibilité d'éditer les sources dans les meilleures conditions possibles	fort
EX_28 : Pouvoir gérer le plan de version sous une gestion de configuration	fort
EX_29 : Pouvoir gérer les recettes dans la gestion de configuration	fort
EX_30 : Pouvoir gérer les outils sous gestion de configuration	fort
EX_31 : Pouvoir gérer les sources des différents composants logiciel sous gestion de configuration	fort
EX_32 : Les sources du projet ne constituent pas une arborescence unique au sein de la gestion de configuration. (chaque outil, chaque composant logiciel, chaque recette, chaque plan de version sont dans des modules de gestion de configuration différents)	fort
EX_33 : L'ensemble du firmware doit pouvoir être généré	fort
EX_34 : Un outil ou un composant logiciel doit pouvoir être compilé de façon unitaire	fort
EX_35 : La génération doit s'arrêter sur une erreur bloquante	fort
EX_36 : l'outil doit aider l'utilisateur à exploiter les erreurs de générations	fort
EX_37 : l'outil doit pouvoir fabriquer un fichier de log des erreurs	fort
EX_38 : Pouvoir relancer une génération de firmware en ne compilant que ce qui est strictement nécessaire	fort
EX_39 : Pouvoir relancer une génération de module en ne compilant que ce qui est strictement nécessaire	fort
EX_40 : rapidité : il doit être possible d'exploiter les processeurs multi-cores pour accélérer la génération.	fort
EX_41 : Pouvoir effacer simplement tous les fichiers générés lors de la génération	fort
EX_42 : Pouvoir effacer simplement tous les fichiers générés d'un module	fort
EX_43 : Pouvoir effacer simplement tous les fichiers installés dans le pseudo filesystem de sortie	moyen
EX_44 : L'IDE doit pouvoir appeler un script ou une application avec les arguments nécessaires pour provoquer le téléchargement du firmware sur la cible.	fort
EX_45 : Pouvoir déboguer des applications	fort
EX_46 : Pouvoir déboguer au niveau du kernel	fort

EX_ 47 : Pouvoir mettre en œuvre facilement le simulateur de Xenomai	moyen
EX_ 48 : Avoir une vue sur les événements internes de la machine	fort
EX_ 49 : Avoir une vue sur l'enchaînement des tâches	fort
EX_ 50 : Avoir une vue sur les appels systèmes	fort
EX_ 51 : Avoir des chronogrammes avec changement d'échelle	fort
EX_ 52 : Pouvoir détecter les mauvaises utilisations de mémoire	fort
EX_ 53 : Pouvoir faire du profiling	fort
EX_ 54 : Pouvoir mesurer l'utilisation de la mémoire	fort
EX_ 55 : Pouvoir faire des mesures de gigue et latence	fort
EX_ 56 : Pouvoir mesurer la charge du CPU	moyen
EX_ 57 : Pouvoir mesurer le débit au niveau des interfaces	moyen
EX_ 58 : Il doit être possible de collecter des informations sur les modules et outils référencés par le plan de version : nom du module, version logicielle utilisée, licence	moyen

6 - Cas d'utilisations de l'environnement et outils de développements

Il est entendu par la suite que le projet RTEL4I s'articule autour d'un générateur de firmware et de l'environnement de développement intégré Eclipse enrichi de plugins (CDT, ...).

Il est convenu par la suite qu'un générateur de firmware référence une collection de recettes qui elles même référencent des modules. Un module est un ensemble de sources logicielles cohérentes qui inclut en son sein les instructions pour sa construction. Ces instructions seront exploitées par les recettes lors de la construction.

Le générateur de firmware est auto suffisant. Eclipse et ses plugins sont donc uniquement des éléments d'interfaces graphiques des fichiers et des outils de développement et de production impliqués dans la construction du firmware. L'ensemble des sources et des outils seront donc scriptables (accessibles en ligne de commande shell) afin d'automatiser la construction d'un firmware et de garantir sa reproductibilité.

Les cas d'utilisations sont structurés en cinq parties:

- **Édition**
Cette partie traite des aides apportées par l'IDE dans l'édition de fichiers sources.
- **Génération**
Cette partie traite des outils de construction et du générateur de firmware.
- **Gestion de configurations**
Cette partie décrit les besoins de gestion de configuration pour ce projet
- **Debug**
Cette partie traite du debug d'applications, de code noyau et des outils associés
- **Test**
Cette partie traite du test des logiciels qui composent le firmware.

Chaque cas d'utilisation décrit son objectif, ses conditions préalables et les résultats attendus.

7 - Un use case typique

Ce chapitre décrit les divers aspects d'un cas d'utilisation dans ce document.

But

Décrit de manière concise le but de ce cas d'utilisation

description

Une description plus détaillée du cas d'utilisation

pré conditions

Une description des conditions nécessaires pour la réalisation de ce cas

trigger

Une description du fait générateur de ce cas (qu'est ce qui le déclenche)

scénario

Une description du scénario idéal de ce cas d'utilisation

scénario alternatif

Éventuellement un ou plusieurs scénarios alternatifs

résultats attendus

Une description des résultats attendus. Des cas particuliers peuvent être décrits pour les scénarios alternatifs

8 - Description des uses cases

8.1 - Outils d'édition

8.1.1 - Choix de l'éditeur

but

Permettre à l'utilisateur de choisir son éditeur favori.

Description

L'adoption de l'IDE passe par un accès facile aux outils génériques avec lesquels les utilisateurs ont construits des aides aux développement logiciel efficaces.

pré conditions

Plusieurs éditeurs sont disponibles sur la machine hôte (gvim, emacs, ...).
L'utilisateur référence son éditeur favori dans l'espace de configuration de l'IDE.

scénario

L'utilisateur essaye d'ouvrir un fichier source de son projet

résultats attendus

L'ouverture d'un fichier depuis l'IDE se fait dans l'éditeur favori de l'utilisateur.

8.1.2 - Édition de sources avec syntaxe colorée

But

Colorer les mots clés en fonction du langage de programmation.

Description

La colorisation syntaxique permet à l'utilisateur de mieux visualiser plus facilement la structure de son programme.

pré conditions

Les sources sont codés selon divers langages de programmations: C, C++, Java, Python, Perl, cpp, m4, Autoconf, Automake: configure.in, configure.ac, makefile.in, makefile.am

scénario

L'utilisateur charge des pages dont le langages est susceptible d'être coloré

résultats attendus

Au sein de l'IDE les mots clés du langage et les commentaires sont colorés différemment du reste du code du fichier en édition.

8.1.3 - Navigation dans le code source

but

Naviguer facilement dans le code d'un projet

Description

L'ensemble du code réparti dans différents fichiers est visualisé avec un minimum de manipulations. L'utilisateur accélère son parcours du code et visualise plus facilement la structure des programmes.

pré conditions

L'ensemble des codes référencés dans les sources d'un module sont présents sur la machine hôte. La base de données associée aux codes sources d'un module est construite par l'IDE (cscope, ctags, ...)

scénario

L'utilisateur sollicite la construction de la base de données associée au source.

A partir d'un fichier de son projet ou de menus dédiés, l'utilisateur navigue dans le code.

résultats attendus

L'utilisateur demande la recherche de la définition et des utilisations d'un mot sélectionné dans le code édité. Présentation dans l'IDE d'une liste comprenant la définition et les utilisations d'un type, d'une classe, d'une variable, d'une macro ou d'une fonction. La définition est distinguée des utilisations.

L'utilisateur demande la liste des fonctions appelées par la fonction pointée: la liste est produite dans l'IDE.

L'utilisateur demande la liste des variables manipulées par la fonction pointée: la liste est produite dans l'IDE.

8.1.4 - Collectes des licences

but

Avoir une vue rapide des licences utilisées par les modules

Description

Afin de pouvoir facilement faire une synthèse sur les licences et la provenance de ses composants logiciel, il est souhaitable que l'utilisateur dispose d'un outil permettant de collecter ces informations dans chacun des modules référencé par le plan de version.

Il s'agit de collecter auprès des outils et des modules une indication de la licence qui est mise en œuvre. Cette information peut alerter le développeur d'un éventuel conflit.

Scénario

L'utilisateur demande dans l'IDE un état des lieux des licences associées au plan de version

résultats attendus

L'outil présente un tableau ou pour chaque entité du plan de version apparaît le nom du module, sa version et la ou les licences sous lesquelles il est distribué.

8.2 - Génération

8.2.1 - Configuration du plan de version

8.2.1.1 - Modification du plan de version : description de la version logicielle

But

Pouvoir choisir une politique de nom de version logicielle

Description

La version logicielle identifie le logiciel généré. L'IDE peut être configuré pour générer

- soit une version officielle
- soit une version de développement

La fabrication du texte de la version peut être confié à un script fourni par l'utilisateur ou rentré manuellement.

pré conditions

Un des modules dans le plan de version doit exploiter le nom de la version logicielle pour ses besoins internes

Scénario

L'utilisateur choisit un des types de version logicielle. Il rentre éventuellement une information manuelle et lance la génération.

résultats attendus

A l'issue de la génération, les modules utilisant la version logicielle l'ont effectivement exploité.

8.2.1.2 - Modification du plan de version : choix de la cible et de son architecture

But

Pouvoir choisir les caractéristiques matérielles de la cible

Description

Un plan de version décrit un logiciel dédié à un certain hardware. Ce cas d'utilisation permet de décrire dans le plan de version les caractéristiques matérielles de la cible

scénario

L'utilisateur remplit dans l'IDE les caractéristiques de la cible. (cela peut se faire avec des menu déroulants proposant les différents types de processeurs et les différentes machines – l'idéal est que ces menus puissent être enrichis par l'utilisateur)

résultats attendus

un plan de version est produit avec les informations hardware de la cible

8.2.1.3 - Modification du plan de version : choix du type de temps Reel

But

Pouvoir choisir le type de temps réel que l'on souhaite avoir : sans TR, avec Xenomai, avec Preempt-RT

Description

Cette option permet à l'utilisateur de choisir en un click l'ensemble des packages nécessaires pour mettre en œuvre un certain type de temps réel

scénario

L'utilisateur choisit dans l'IDE la nature du temps réel

scénario alternatif

L'utilisateur a aussi le choix entre diverses skin xenomai

résultats attendus

Le plan de version est enrichi des composants nécessaires à la mise en œuvre du temps réel choisi

8.2.1.4 - Modification du plan de version : gestion des modules

but

Édition du plan de version afin de le créer ou le modifier.

Description

A tout moment, un développeur doit pouvoir faire des changements de son plan de version et voir ces changements exploitables directement.

scénario

L'utilisateur ajoute, retire des outils ou des composants logiciels. A chacune de ces action, il lance une génération

résultats attendus

A l'issue de la génération, les changements sollicités par l'utilisateur ont été pris en compte.

8.2.2 - Générations

8.2.2.1 - Génération d'un firmware selon son plan de version

but

Générer le firmware décrit par son plan de version.

Description

Le générateur exécute les méthodes des recettes associées à chaque composant du firmware référencé dans le plan de version. Les dépendances inter modules présentes dans les recettes s'exercent afin de garantir le succès de la construction du firmware.

pré conditions

Le plan de version est suffisant pour décrire les outils et les modules pour construire le firmware.

scénario

Depuis l'IDE ou une console shell l'utilisateur exécute la commande du générateur de firmware pour construire le firmware décrit par son fichier plan de version.

résultats attendus

Le générateur produit un fichier binaire suffisant pour démarrer la cible et exécuter l'application visée. Le firmware en exécution sur le système cible peut afficher son nom et son numéro de version. Un ensemble de fichiers de log enregistrent les étapes de la génération et les erreurs éventuelles. La construction du firmware est déterministe et reproductible quelque soit la machine hôte.

8.2.2.2 - Re-Génération d'un firmware selon son plan de version**but**

Re-Générer le firmware décrit par son plan de version alors que des modifications ont été faites

Description

Très proche de la génération de firmware, ce use case décrit ce qui doit se passer lorsqu'un développeur ou un intégrateur amène des modifications aux sources, aux recettes, ou au plan de version

scénario

L'utilisateur fait des modifications aux sources, aux recettes, ou au plan de version

résultats attendus

Le firmware est re-généré en ne recompilant que ce qui est strictement nécessaire.

8.2.2.3 - Génération d'outils de cross compilation**But**

Pouvoir générer de façon unitaire un outil depuis le plan de version

description

Le plan de version décrit les divers outils nécessaires à la génération de firmware. Lors de la construction de son plan de version, le développeur doit pouvoir tester de façon incrémentale la cohérence de celui-ci.

pré conditions

L'outil est décrit dans le plan de version

scénario

L'utilisateur sollicite la génération de cet outil en particulier à partir de son IDE ou à partir d'une commande en ligne depuis un shell.

résultats attendus

L'outil est généré et si la recette le demande est installé dans le répertoire d'installation.

8.2.2.4 - Compilation d'un module**But**

Pouvoir générer, de façon unitaire, un des modules décrits dans le plan de version.

description

Le développeur doit pouvoir solliciter à tout moment la génération d'un module décrit dans un plan de version et obtenir le résultat simplement.

scénario

Depuis l'IDE ou un shell l'utilisateur exécute la commande du générateur de firmware de compilation d'un module.

résultats attendus

Le générateur de firmware assure la génération du module en résolvant éventuellement les dépendances nécessaires en respectant les contraintes du plan de version.

8.2.2.5 - Re-compilation d'un module**But**

Pouvoir relancer à tout moment la génération d'un module

Description

Une fois qu'un développeur a construit un module du firmware. Il peut modifier les sources et solliciter une nouvelle compilation. Il faut alors que seuls les binaires sensibles à ces modifications soient recompilés.

pré conditions

L'utilisateur a déjà compilé un module

scénario

L'utilisateur fait des modifications à des sources de ce module.

résultats attendus

Seuls les fichiers du module devant être recompilés le sont réellement.

8.2.2.6 - Construction d'une recette**but**

Pouvoir construire depuis une recette la plupart des types de module de la communauté

description

La communauté open source produit du code hétérogène. La façon de construire et d'installer ces codes l'est aussi. La recette est un moyen de faire abstraction de cette diversité.

Lorsqu'il construit une recette pour un module de la communauté, il faut que le développeur puisse aborder les modes de construction les plus variés.

- Make
- gmake
- cmake
- autotools
- scons
- kbuild
- ...

La recette doit pouvoir gérer les diverses tâches à accomplir

- récupération des sources (la recette décrit où les sources peuvent être trouvés, y compris ceux des patch)
- patch
- préparation
- compilation
- installation

La recette décrit aussi les éventuelles dépendances par rapport à un outil ou un autre module.

scénario

L'utilisateur construit sa recette en fonction de la technique de construction employée dans la communauté. Il lance la génération du module décrit par sa recette.

résultats attendus

Le module se construit comme si il avait été lancé «à la main »

8.2.2.7 - Parallélisation de la génération

but

Accélérer la construction d'un firmware en parallélisant les constructions intermédiaires.

description

Le générateur distribue les constructions intermédiaires (compilation, expansion) sur plusieurs jobs. Les jobs peuvent être exécutés en local ou sur des machines distantes identifiées au sein du générateur.

pré conditions

La gestion des dépendances au sein d'un module et inter modules est infallible. :-)

résultats attendus

Le firmware généré est identique à celui qui serait construit en locale avec un seul job.

8.2.2.8 - Profils

Ces quelques uses cases sont pertinents dans le cas ou le choix de mettre en œuvre la notion de profils est retenue. Ils montrent divers aspects que peut amener cette notion.

8.2.2.8.1 - Chaque profil a son espace de production**But**

Pouvoir disposer pour chaque profil d'un espace où est envoyé le code généré

description

Il doit pouvoir être possible de disposer de façon simultanée des générations de 2 profils afin de pouvoir passer de l'un à l'autre sans tout régénérer.

pré conditions

Deux profils au moins existent dans un projet.

Scénario

L'utilisateur fait des modifications au niveaux des sources, au niveau des recettes ou au niveau du plan de version. Il lance les générations pour chacun des profils

résultats attendus

Les générations sont bien séparées. Chaque profil a son jeu complet de binaires et son propre firmware

8.2.2.8.2 - Options génériques pour tout les profils**But**

Pouvoir décrire des options qui sont prise en compte par tout les profils

description

Il peut être intéressant de décrire des options qui sont prises en compte par l'ensemble des profils (ne pas avoir à les recopier, profil par profil). Ces options sont alors diffusées à chaque module et outil pour chaque profil.

pré conditions

Deux profils au moins existent dans un projet.

Scénario

L'utilisateur ajoute une option générique et lance les générations de l'ensemble des profils.

résultats attendus

Tout les outils et tout les modules ont pris en compte l'option pour chaque profil.

8.2.2.8.3 - Options spécifiques à un profil

But

Pouvoir décrire des options pour un seul profil

description

Ces options sont alors diffusées à chaque module et outil pour ce profil.

pré conditions

Deux profils au moins existent dans un projet

Scénario

L'utilisateur ajoute une option spécifique à ce profil et lance les générations de l'ensemble des profils.

résultats attendus

Tous les outils et tous les modules ont pris en compte l'option seulement pour ce profil

8.2.2.8.4 - Composition variable d'un firmware

But

Pouvoir avoir plus ou moins de modules et outils dans un firmware en fonction du profil.

description

Des modules ajoutés dans un plan de version sont pris en compte ou invalidés en fonction du profil lors de la génération.

pré conditions

Deux profils au moins existent dans un projet qui a des outils et des modules.

Scénario

L'utilisateur invalide des outils et des modules dans un profil et les laisse dans un autre profil. Il génère les 2 profils.

résultats attendus

Suivant les variables conditions le firmware embarque les bons modules.

8.2.2.8.5 - Génération spécifique d'un module en fonction du profil

But

Pouvoir générer de façon spécifique un module pour un profil donné

description

Ce use case permet de rendre spécifique la génération d'un module pour un profil donné. Cela permettrait par exemple de pouvoir facilement concentrer les actions de debug sur un périmètre réduite de module.

Ce use case suppose alors que la notion de profil s'applique également sur une recette.

Scénario

L'utilisateur se concentre sur un module dans un profil donné. Il change des options et régénère.

résultats attendus

Le module de ce profil voit son binaire produit différent de ceux des autres profils.

8.2.2.9 - Patch d'un module**but**

Appliquer des patches correctifs ou évolutifs au sources codes d'un module noyau.

description

Une recette référence les patches d'un module et leurs ordres d'application. Elle décrit également sur quel dépôt trouver ces patches.

scénario

L'utilisateur lance la génération d'un module dont la recette référence des patches

résultats attendus

Lors de la génération du module, les sources sont patchés dans le bon ordre avant la préparation.

8.2.3 - Informations sur la génération

Ce sont des informations utiles à l'utilisateur lorsqu'il s'agit de résoudre un problème de génération.

8.2.3.1 - Graphe des dépendances d'une construction au sein d'un module**But**

Donner à l'utilisateur des informations sur les fichier réellement inclus dans sa construction.

Description

Dans des environnements complexes, l'utilisateur peut penser inclure un fichier en particulier alors que c'est un autre qui est réellement inclus.

pré conditions

La recette ou les makefiles du module sont configurés de telle manière que, lors de la compilation, les fichiers de dépendance (extension *.d » soient générés

scénario

L'utilisateur sollicite la génération de ces fichiers de dépendance si cela n'est pas fait implicitement. Il consulte un menu de son IDE lui donnant accès à ces dépendances.

résultats attendus

L'utilisateur voit dans son IDE, pour chaque fichier sources, les includes qui sont réellement exécuté lors de la compilation.

8.2.3.2 - Graphe des dépendances d'un module

But

Montrer à l'utilisateur le graphe de dépendance inter modules. Montrer les modules manquants dans le plan de version.

description

Cet outil permet de montrer les diverses dépendances exprimées par chacun de ses outils et modules. Ces informations de dépendances sont celles qui sont décrites dans les recettes

scénario

L'utilisateur demande pour un plan de version donné la liste des dépendances dans un menu de l'IDE.

résultats attendus

L'arbre de dépendances du module désigné est affiché dans l'IDE. En rouge sont montrées les dépendances présentes dans les recettes mais non décrites dans le plan de version. (on peut imaginer qu'une aide à l'insertion rapide des modules manquants dans le plan de version soit une fonction de l'IDE. Ce qui permettrait de construire facilement des plans de version officiel).

8.2.3.3 - Exploiter les erreurs de construction

But

Exploiter au mieux les messages d'erreur lors de la génération.

description

Toutes les erreurs tracées par le générateur de firmware sont accessibles depuis l'IDE.

Une colorisation du texte peut marquer les messages sur divers critères (présence des mots errors/warning ou provenance du message gcc, make)

Un simple clic sur un message doit permettre, si cela est possible, de synchroniser le fichier texte correspondant et montrer la ligne en cause.

scénario

L'utilisateur lance la génération du soft. Lors de la génération, une fenêtre de l'IDE apparaît dans laquelle défile les messages d'erreur. L'utilisateur peut alors cliquer sur un des messages d'erreur

résultats attendus

Sur chaque clic de l'utilisateur, l'éventuel fichier source en cause apparaît dans une fenêtre avec une indication de la ligne d'erreur sous la forme d'un marqueur.

8.2.3.4 - Création d'un fichier de log des erreurs de construction

But

Exploiter au mieux les messages d'erreur lors de la génération.

description

Une configuration de l'IDE permet de tracer les erreurs créées par le générateur de firmware dans un fichier de log.

pré conditions

L'utilisateur a configuré son IDE de façon à créer un fichier de log contenant les erreurs.

scénario

L'utilisateur lance la génération du soft.

résultats attendus

Lors de la génération, le fichier de log est créé.

8.3 - Gestion de Configurations à partir de l'IDE

Il est important de noter que les plans de versions, les recettes et les sources ne sont pas dans la même arborescence de source.

Pour garantir l'indépendance de ces divers concepts, ils doivent être hébergés sur des sites de gestion de configuration séparés.

8.3.1 - Gestion du plan de version

But

Gérer les plans de versions dans la gestion de configuration.

Description

Les plans de versions doivent pouvoir être gérés comme des sources dans un outil de gestion de configuration depuis l'IDE.

Scénario

L'utilisateur utilise la gestion de configuration pour gérer son plan de version.

résultats attendus

Depuis l'IDE, l'utilisateur peut importer son plan de version dans un système de gestion de configuration, le taguer, faire des branches, des merges, des diff, voir les historiques, voir le status

8.3.2 - Gestion des recettes

But

Gérer les recettes dans la gestion de configuration.

Description

Les recettes doivent pouvoir être gérées comme des sources dans un outil de gestion de configuration depuis l'IDE.

Scénario

L'utilisateur utilise la gestion de configuration pour gérer ses recettes.

résultats attendus

Depuis l'IDE, l'utilisateur peut importer ses recettes dans un système de gestion de configuration, le taguer, faire des branches, des merges, des diff, voir les historiques, voir le status

8.3.3 - Gestion des sources**But**

Gérer les sources dans la gestion de configuration.

Description

Les sources doivent pouvoir être gérées dans un outil de gestion de configuration depuis l'IDE.

Scénario

L'utilisateur utilise la gestion de configuration pour gérer ses sources.

résultats attendus

Depuis l'IDE, l'utilisateur peut importer ses sources dans un système de gestion de configuration, le taguer, faire des branches, des merges, des diff, voir les historiques, voir le status

8.4 - Debug

Le debug peut être une tâche délicate sur l'embarqué si il n'a pas été préparé lors de la construction du logiciel. Il doit pouvoir être mis en place facilement pour être utilisé par le développeur. La non utilisation du debugger est souvent lié à des difficultés de mise en œuvre.

8.4.1 - Debug d'applications**but**

Pouvoir synchroniser une application d'un embarqué avec un debugger.

Description

Les applications embarquées sont debugués en général avec un couple gdb/gdb-server. Suivant les contraintes, elles peuvent être testée sur :

- un système émulé (QEMU)
- directement sur la machine embarquée
- sur la machine embarquée avec l'application à déboguer sur un filesystem externe.

Il s'agit d'ouvrir une session gdb sur la machine hôte de façon à ce qu'elle se synchronise avec une application à debugger sur la machine cible en s'appuyant sur un gdb server. Gdb et gdbserver communiquent via un port série ou via le réseau sur TCP ou UDP.

scénario

L'utilisateur construit depuis un shell ou l'IDE l'application avec les symboles de debug. Gdbserver est ajouté au plan de version (profil debug) firmware et configuré pour exécuter

l'application erronée. Il ouvre ensuite la session gdb sur le système hôte et se connecte via le port tcp au gdbserver en exécution sur cible.

scénario alternatif

L'utilisateur débogue son application dans une machine émulée.

résultats attendus

L'utilisateur peut arrêter l'application, poser des breakpoints, exécuter le code pas à pas, éditer la mémoire, les variables globales ou la pile, éditer les registres processeur, basculer d'un thread à l'autre, le programme counter courant est synchronisé avec les sources C/C++ dans une fenêtre de l'IDE.

8.4.2 - Debug de kernel

Le debug du kernel avec synchronisation des sources suppose que le noyau ait été compilé avec les options adéquates. Deux versions d'images doivent être alors générées :

Une version avec les symboles exploitable par un débogueur ou un émulateur et une version sans symbole (strippée) embarquable sur le produit.

Il est impératif de pouvoir choisir les arguments passés au noyau au démarrage.

8.4.2.1 - Debug dans le kernel par KGDB ou kdb

but

Pouvoir se synchroniser avec du code du kernel ou d'un module kernel

Description

Un agent kgdb ou kdb est inclus dans l'image du kernel lors de sa recompilation. Le kernel peut alors être debuggé avec un simple gdb connecté au travers d'un lien série ou d'une connexion réseau.

Scénario

L'utilisateur configure son IDE pour solliciter l'inclusion d'un agent kgdb dans le kernel. Une fois son firmware incluant kgdb ou kdb construit et installé sur la cible, il demande à son IDE de démarrer une session debug se synchronisant avec sa cible.

résultats attendus

L'utilisateur peut arrêter la machine distante par kgdb ou kdb, poser des points d'arrêt, dans le code du kernel ou dans le code des modules kernel. Comme pour le debug d'application, l'utilisateur voit son source C/C++ dans la fenêtre de son IDE et dispose de toutes les fonctionnalités de son débogueur.

Les kernel panic sont capturés par kgdb ou kdb

8.4.2.2 - Debug sur machine émulée Qemu

But

Pouvoir se synchroniser avec du code du kernel ou d'un module kernel d'une machine émulée

description

Cette technique de debug s'appuie sur un stub synchronisable avec un gdb de façon à pouvoir débogger facilement le kernel d'une machine émulée.

pré conditions

Disponibilité de la machine virtuelle pour l'architecture visée.

Scénario

L'utilisateur configure son IDE de façon à pouvoir générer un firmware et disposer des outils QEMU nécessaires

résultats attendus

Le firmware est émulé sur une plateforme linux.

8.4.2.3 - Debug du noyau avec Kprobes**but**

Pouvoir debugger dans le kernel en utilisant la technique des kprobes

description

Kprobes permet de définir des fonctions prologue et épilogue à l'exécution d'une adresse donnée du noyau. Ceci permet donc de tracer l'exécution du noyau en des points précis comme l'adresse d'un kernel panic par exemple.

scénario

Dans un module noyau dédié l'utilisateur définit une fonction prologue (resp épilogue) à une fonction noyau choisie qu'il enregistre dans la structure de donnée jprobes (resp kretprobes). Si besoin il enregistre un prologue et un épilogue à une adresse d'exécution choisie dans la structure de donnée kprobe. Il compile le module et l'insère au moment opportun dans le noyau en exécution.

résultats attendus

Le module inséré produit les informations de debug escomptées à l'exécution des adresses enregistrées dans les probes.

8.4.2.4 - Utilisation de support hardware au travers du JTAG**but**

Debugger le système en préservant l'application de l'utilisation des interfaces génériques usb, ethernet.

Description

Il est possible d'avoir sur certain hardware embarqué une sortie jtag La synchronisation avec le noyau est alors possible en utilisant un émulateur hardware (par exemple une sonde lauterbach).

pré conditions

Une application qui interface le câble jtag.

Scénario

L'utilisateur indique dans l'IDE qu'il veut debugger avec un emulateur

L'utilisateur ouvre une session gdb, établie une connexion tcp avec le serveur gérant le jtag et procède au debug de son application.

résultats attendus

Utilisation de gdb ou autre pour faire un debug complet dans le kernel.

8.4.2.5 - Debug du noyau avec un dump lkcd**but**

Exploiter une image de la mémoire obtenue lors d'un crash du système.

description

La construction du dump mémoire dans un format exploitable se fait en deux étapes. Lorsque le système crash l'image mémoire est sauvée sur une partition dédiée ou la partition de swap, ensuite le redémarrage est forcé. La deuxième étape relit l'image mémoire et l'archive dans une arborescence finale par exemple /var/log/dump/ avec d'autres fichiers contenant la table des symboles, les structures de données utilisées par le kernel (kerntypes), une analyse textuelle du crash.

pré conditions

La cible dispose d'un filesystem RW pour accueillir le fichier dump

scénario

Le développeur utilise son IDE pour lire le fichier dump en s'appuyant sur les outils lkcdutils et lcrash permettent d'analyser l'image mémoire produite.

résultats attendus

Les divers fichiers d'analyse de crash sont produits dans des fenêtres eclipse : dump.n, kerntypes.n, map.n, analysis.n, lcrash.n

8.4.2.6 - Debug avec le simulateur Xenomai**But**

Mettre en œuvre le simulateur Xenomai et faire tourner dedans nos applications.

Description

Il s'agit d'exploiter au mieux le simulateur Xenomai dans l'IDE. Les difficultés de mise en œuvre devront être cachées à l'utilisateur afin que celui ci puisse se concentrer sur le fond du problème : le temps réel

scénario

Depuis l'IDE l'utilisateur demande l'exécution du firmware dans Xenosim.

résultats attendus

L'utilisateur visualise dans l'IDE l'exécution du firmware.

8.5 - Analyse

8.5.1 - Analyse générique d'événements

but

Tracer le flot d'événements qui s'exerce au sein du système sur une période donnée.

description

Ce use case permet au développeur de faire une analyse de l'enchaînement des divers élément. Un chronogramme présente avec des couleurs différentes l'ensemble des tâches et des interruptions qui s'exercent au sein du système. Il est possible de visualiser les temps associés aux réponses aux interruptions (latence, gigue), aux changements de contexte entre processus ou thread (incluant les changements de domaine xenomai), aux appels système. Il est possible de changer l'échelle de temps afin de se focaliser sur un petit groupe d'événements ou d'en visualiser un plus grand nombre. Une synthèse par processus est accessible, elle détaille le temps passé dans l'espace utilisateur, dans l'espace noyau, dans les changements de contexte, dans les handlers d'interruption, dans l'attente d'un verrou et d'une opération d'entrée/Sortie, les appels systèmes réalisés. Des exemples de graphiques pertinents peuvent être trouvés sur le site de opersys (<http://www.opersys.com/LTT/screenshots.html>)

scénario

Depuis l'IDE l'utilisateur demande l'enregistrement des traces du firmware en exécution pour une période qu'il spécifie. Il demande ensuite à visualiser le chronogramme.

résultats attendus

Le firmware est généré suivant le profil trace de son plan de version et exécuté sur cible. Le chronogramme est chargé dans l'IDE, l'utilisateur accède à l'ensemble des fonctionnalités décrite ci-dessus.

8.5.2 - Analyse des événements avec LTT

but

Tracer le flot d'événements qui s'exerce au sein du système sur une période donnée.

description

Un module noyau associé à un processus démon permet d'enregistrer sur disque un ensemble d'événements afin d'analyser les interactions entre le noyau, les applications utilisateur et plus généralement entre les différentes tâches du système. Il est alors possible de visualiser avec des outils spécifiques le graphe des événements, l'activité d'un processus afin d'appréhender le respect des contraintes temps réel et la performance du système.

pré conditions

L'embarqué a accès à une mémoire de masse conséquente.

scénario

Depuis l'IDE, l'utilisateur demande à exploiter LTT et génère le firmware

résultats attendus

L'outil configure le plan de version pour mettre en oeuvre LTT.

Sur l'embarqué, le démon de traces démarre et produit les enregistrements pour la période visée dans le système de fichier cible. L'outil de visualisation exploite les fichiers de traces produits par le démon afin de présenter dans un chronogramme pour la période d'enregistrement les processus en exécution, les changements de contexte entre processus et thread. L'utilisateur visionne pour chaque processus l'ensemble des appels système, les signaux, les traps, les interruptions matérielles et logicielles, ses interactions avec le noyau et les autres processus. Les taches xenomai sont naturellement intégrées dans les traces produites sans traitement particulier de l'utilisateur.

8.5.3 - Analyse de performance avec Oprofile**but**

Localiser les codes pour lesquels le cpu consomme le plus de temps.

description

Un module noyau et un processus démon produisent des traces du système en exécution. Les traces enregistrent les temps d'exécution au sein du noyau, des modules noyau, des handlers d'interruption, des bibliothèques partagées et des applications. Les événements tracés par les registres processeur comme le comptage des cycles CPU, des cache miss sont également intégrés dans les traces oprofile.

pré conditions

L'image vmlinux non compressée doit être disponible pour tracer le noyau. Le code à tracer est construit normalement sans option de profiling ajoutée. Pour avoir le graphe des appels un patch est requis mais seul la plateforme x86 est supportée.

scénario

L'utilisateur compile le module noyau oprofile, installe le processus démon et les utilitaires associés. Une fois le module oprofile chargé dans le noyau en exécution, oprofile est configuré puis démarré (opcontrol --start). Les applications à tracer sont exécutées puis les traces sont périodiquement produites dans un répertoire cible (opcontrol --dump)

résultats attendus

L'outil de synthèse des traces oprofile (oreport) présente la répartition des codes exécutés suivant les métriques sélectionnés à la configuration. La configuration et l'exploitation des traces sont accessibles depuis une console texte ou l'IDE. Dans l'IDE les traces et les fichiers sources sont corrélés.

8.5.4 - Analyse d'utilisation de la mémoire avec Valgrind**but**

Détecter les mauvaises utilisations de la mémoire.

description

L'outil memcheck de la suite valgrind permet de signaler les mauvais accès des zones mémoires d'un processus, les utilisations de variables avant leur initialisation, les libérations de mémoire excessives et plus généralement les fuites mémoires.

pré conditions

L'application à analysée doit être compilée en mode debug pour lier les lignes du code source avec les adresses du programme tracées par l'outil.

scénario

Le firmware embarque l'outil memcheck ou il est accédé depuis un point de montage NFS. L'application memcheck est exécutée avec le programme à analyser.

résultats attendus

Les traces sont produites dans une console texte ou l'IDE. Dans l'IDE les traces et les codes sources sont corrélés.

8.5.5 - Exploration statique d'un binaire**but**

Visualiser les caractéristiques d'un fichier binaire depuis l'IDE.

description

Les outils de la chaîne croisée sont accessibles depuis l'IDE pour présenter le désassemblé d'un binaire, le mapping mémoire, les tailles des sections.

scénario

L'utilisateur appelle depuis l'IDE les outils de binutils objdump, nm, readelf, size, etc ...

résultats attendus

La sortie produite par les outils binutils est présentée dans l'IDE.

8.5.6 - Analyse des interactions d'une application avec le système**but**

Tracer les interactions d'une application désignée avec le système.

description

Les appels aux bibliothèques dynamiques, les appels systèmes et les signaux reçus sont enregistrés afin de visualiser les interactions de l'application visée avec le reste du système.

scénario

L'utilisateur demande à l'IDE de mettre en œuvre ltrace, l'outil configure le plan de version, construit et embarque le programme ltrace dans le firmware. L'application visée est simplement exécutée depuis ltrace.

résultats attendus

Les appels systèmes, les appels aux bibliothèques dynamiques réalisés par l'application visée et les signaux qu'elle reçoit sont enregistrés dans un fichier système cible et présentés dans l'IDE.

8.5.7 - Analyse des statistiques d'ordonnement avec schedtop**but**

Visualiser les statistiques d'ordonnement.

description

Les statistiques produites par le noyau dans les fichiers /proc/schedstat /proc/pid/schedstat sont présentés afin de visualiser l'ordonnement des processus du système.

scénario

L'utilisateur demande à l'IDE d'utiliser schedtop. L'outil configure le plan de version, construit et embarque l'utilitaire schedtop dans le firmware.

résultats attendus

Les statistiques d'ordonnement sont présentées dans l'IDE. (ou dans le shell de la busybox)

8.5.8 - Analyse des applications multi thread avec PTT**but**

Visualiser les interactions d'un processus multi thread avec la bibliothèque POSIX Thread.

description

Trace l'exécution des threads de l'application afin de présenter dans le temps les événements associés aux objets POSIX, mutex, sémaphores, variables conditions, queues de messages, etc...

scénario

L'utilisateur demande depuis l'IDE de générer l'analyse d'une application désignée pour la période qu'il spécifie.

résultats attendus

Le firmware est chargé et exécuté sur cible. Les traces produites sont visualisées dans l'IDE.

8.5.9 - Analyse des statistiques d'ordonnement**but**

Visualiser les statistiques d'ordonnement pour l'ensemble des tâches d'un système sur une période donnée.

description

Tracer le nombre de commutations de contexte, de préemption, de blocage sur ressources partagées.

scénario

Depuis l'IDE l'utilisateur demande les statistiques d'ordonnement pour une durée.

résultats attendus

Le firmware est construit puis exécuté sur cible. Les statistiques sont présentées dans l'IDE.

8.6 - Mesure**8.6.1 - Taux de couverture de code****but**

Mesurer quelles parties du code d'un module ne sont pas testées.

description

Pour chaque test réalisé sont présentées les lignes de code du module qui sont exécutées avec leur fréquences et celles qui ne le sont pas. Il est ainsi possible de mesurer la proportion de code testé et les parties du code les plus sollicitées.

pré conditions

Disposition d'un ensemble de tests unitaires du module

scénario

Depuis le shell ou l'IDE, l'utilisateur demande la génération du firmware avec la construction du module désigné adaptée au test de couverture. Il lance l'application a mesurer

résultats attendus

Pour chaque fichier source du module le pourcentage de ligne de code couverte par le test est présenté dans l'IDE. Le détail par ligne de code est accessible. Le pourcentage global produit par l'ensemble des tests du plan est affiché.

8.6.2 - Mesure de latence et gigue**but**

Mesurer les latences du code et leur gigue pour une période donnée.

Description

L'utilisateur designe les point à mesurer. Sont présentés les temps de latence mesurés, la moyenne et la gigue résultante.

scénario

Depuis l'IDE l'utilisateur demande la mesure de gigue et latences pour un code particulier

résultats attendus

Une mesure est faite. L'ensemble des latences mesurées, les moyennes et les giges sont présentées dans l'IDE.

8.6.3 - Mesure de débit

but

Mesurer les flux de données au sein du système.

Description

Mesurer les débits sur les interfaces du système, port usb, port ethernet., port ip, port wan.

scénario

Depuis l'IDE l'utilisateur demande une mesure de débit et spécifie les caractéristiques des ports à mesurer.

résultats attendus

Le firmware est éventuellement construit puis exécuté sur cible. L'utilisateur visualise les débits dans l'IDE.

8.6.4 - Mesure de la charge du système

but

Contrôler la charge CPU dans le temps.

description

Visualisation du graphe de charge du CPU pour une période donnée.

scénario

Depuis l'IDE l'utilisateur définit la période et la fréquence d'échantillonnage pour la mesure.

résultats attendus

Le firmware est éventuellement construit et exécuté sur cible puis l'utilisateur visualise le graphe dans l'IDE.

8.6.5 - Mesure de la mémoire

but

Mesurer l'utilisation de la mémoire dans le firmware.

description

Présenter sur une période donnée l'utilisation de la mémoire pour chaque processus.
Présentation des tailles de mémoire résidente, de mémoire partagée, de pile, du nombre de pages demandées, de page fault, de swap.

scénario

Depuis l'IDE l'utilisateur demande de mesurer l'utilisation de la mémoire pour une période qu'il spécifie.

résultats attendus

Le firmware est éventuellement construit et exécuté sur cible. Les mesures sont présentées par processus dans l'IDE.

9 - Notation des cas d'utilisation

Ce paragraphe décrit l'importance du besoin associé à chacun des cas d'utilisation

Nom du cas d'utilisation	besoin
8.1.1 - Choix de l'éditeur	moyen
8.1.2 - Édition de sources avec syntaxe colorée	fort
8.1.3 - Navigation dans le code source	fort
8.1.4 - Collectes des licences	moyen
8.2.1.1 - Modification du plan de version : description de la version logicielle	fort
8.2.1.2 - Modification du plan de version : choix de la cible et de son architecture	fort
8.2.1.3 - Modification du plan de version : choix du type de temps Reel	fort
8.2.1.4 - Modification du plan de version : gestion des modules	fort
8.2.2.1 - Génération d'un firmware selon son plan de version	fort
8.2.2.2 - Re-Génération d'un firmware selon son plan de version	fort
8.2.2.3 - Génération d'outils de cross compilation	fort
8.2.2.4 - Compilation d'un module	fort
8.2.2.5 - Re-compilation d'un module	fort
8.2.2.6 - Construction d'une recette	fort
8.2.2.7 - Parallélisation de la génération	fort
8.2.2.8.1 - Chaque profil a son espace de production	fort
8.2.2.8.2 - Options génériques pour tout les profils	fort
8.2.2.8.3 - Options spécifiques à un profil	fort
8.2.2.8.4 - Composition variable d'un firmware	fort
8.2.2.8.5 - Génération spécifique d'un module en fonction du profil	fort
8.2.3.1 - Graphe des dépendances d'une construction au sein d'un module	moyen
8.2.3.2 - Graphe des dépendances d'un module	moyen
8.2.3.3 - Exploiter les erreurs de construction	fort
8.2.3.4 - Création d'un fichier de log des erreurs de construction	fort
8.3.1 - Gestion du plan de version	fort
8.3.2 - Gestion des recettes	fort
8.3.3 - Gestion des sources	fort
8.4.1 - Debug d'applications	fort
8.4.2.1 - Debug dans le kernel par KGDB ou kdb	fort

8.4.2.2 - Debug sur machine émulée Qemu	fort
8.4.2.3 - Debug du noyau avec Kprobes	moyen
8.4.2.4 - Utilisation de support hardware au travers du JTAG	fort
8.4.2.5 - Debug du noyau avec un dump lkcd	fort
8.4.2.6 - Debug avec le simulateur Xenomai	fort
8.5.1 - Analyse générique d'événements	fort
8.5.2 - Analyse des événements avec LTT	fort
8.5.3 - Analyse de performance avec Oprofile	fort
8.5.4 - Analyse d'utilisation de la mémoire avec Valgrind	fort
8.5.5 - Exploration statique d'un binaire	moyen
8.5.6 - Analyse des interactions d'une application avec le système	fort
8.5.7 - Analyse des statistiques d'ordonnancement avec schedtop	moyen
8.5.8 - Analyse des applications multi thread avec PTT	fort
8.5.9 - Analyse des statistiques d'ordonnancement	fort
8.6.1 - Taux de couverture de code	moyen
8.6.2 - Mesure de latence et gigue	fort
8.6.3 - Mesure de débit	fort
8.6.4 - Mesure de la charge du système	fort
8.6.5 - Mesure de la mémoire	fort