



open wide
INGENIERIE

Validation des outils

Version: 1.0

Auteur: Pierre Ficheux

- PROJET RTEL4I -

RTEL4i
REAL TIME EMBEDDED LINUX FOR INDUSTRIES

Validation des outils (L1.3-b)


ESGER3

 **open wide**
INGENIERIE


Mandriva

SAGEMCOM




	Validation des outils	Version: 1.0 Auteur: Pierre Ficheux
---	-----------------------	--

MODIFICATIONS

VERSION	DATE	AUTEUR(S)	DESCRIPTION
1.0	10/2/2011	P. Ficheux	Création

Table des matières

1.Présentation du document et du contexte.....	4
2.Outils concernés.....	4
3.Détail des outils.....	4
3.1.Chaînes de compilation.....	4
3.1.1 ARM.....	5
3.1.2 SH4.....	5
3.1.3 Altera NIOS II.....	5
3.1.4 Xilinx Microblaze.....	5
3.2.Outil de production Buildroot.....	6
4. Test de production du firmware.....	6
4.1. Test de validation de compilation.....	6
4.2. Test d'exécution sur QEMU.....	7
4.3. Test sur cible réelle, utilisation du « virtual lab » (VLAB).....	8

	<p>Validation des outils</p>	<p>Version: 1.0 Auteur: Pierre Ficheux</p>
---	------------------------------	--

Index des illustrations

Figure 1. Console du VLAB, switch et MOXA.....10
 Figure 2. Les cibles SH4, Screen et FemtoCell sur le VLAB.....11

1. Présentation du document et du contexte

Le but de ce document est de décrire les méthodes de validation des outils adaptés développés lors du SP1.3 et concernant l'enrichissement des chaînes de compilation.

2. Outils concernés

Les types d' outils adaptés durant le SP1.3 sont les suivants :

1. Compilateur (GCC)
2. Bibliothèques de développement: Glibc, uClibc, etc. I
3. Débogueurs et agents de mise au point: GDB, GDBSERVER, KGDB
4. Outils de mise au point spécifiques: émulateur, simulateur événementiel pour le temps réel
5. Outil de productions de distributions. Ces outils permettent d'intégrer les composants précédents dans un ensemble de scripts permettant d'automatiser la production du firmware de la cible en fonction des choix de l'utilisateur.

Dans la plupart des cas, les composants cités aux point 1, 2 et 3 sont rassemblés sous la dénomination de *chaîne de compilation*. Par définition, la chaîne de compilation permet :

1. De produire du code assembleur de la cible à partir du langage évolué (C/C++), c'est la tâche de la partie compilation GNU-cc (gcc).
2. De produire de fichiers objets à partir du fichier assembleur, c'est la tâche du programme d'assemblage GNU-as (gas)
3. De produire un exécutable à partir des fichiers objets et des bibliothèques présentes dans la chaîne de compilation (au minimum la libc).

3. Détail des outils


Dans ce paragraphe, nous allons décrire en détail les composants utilisés.

3.1. Chaînes de compilation

Ces chaînes sont utilisées pour toutes les nouvelles architectures traitées par RTELI, soit :

- ARM (plusieurs versions avec cœurs ARM9 ou ARM11)
- SH4
- Altera NIOS II avec ou sans MMU
- Xilinx Microblaze

Ces compilateurs étant tous issus du projet GNU et donc diffusés sous GPL, il est toujours possible de produire les binaires à partir des sources, même c'est peu fréquent.

	Validation des outils	Version: 1.0 Auteur: Pierre Ficheux
---	-----------------------	--

3.1.1 ARM

Nous avons utilisé le plus fréquemment les chaînes de compilation éditées par la société *Code Sourcery* (<http://www.codesourcery.com>). Ces produits sont déjà *validés* pour des utilisations industrielles et existent également sous forme « Lite » (sans support officiel) du fait de la licence GPL utilisée qui oblige la redistribution des sources.

Plusieurs versions ont été testées au cours de l'évolution du projet (la version 2010.09 étant la plus utilisée).

- Version 2008q3 sur <http://www.codesourcery.com/sgpp/lite/arm/portal/release644>
- Version 2010.09 sur <http://www.codesourcery.com/sgpp/lite/arm/portal/release1600>
- Version 2011.03 sur <http://www.codesourcery.com/sgpp/lite/arm/portal/release1803>

Nous avons également utilisé au début du projet un compilateur croisé construit avec *Crosstool-NG*, cet outil étant disponible sur <http://crosstool-ng.org>. Cependant cet outils nécessite une configuration assez complexe et peut poser des problèmes au niveau de la compatibilité entre les différents composants de la chaîne: compilateur GCC, libc, noyau Linux. Nous utilisons désormais Code Sourcery.

3.1.2 SH4

Le support du SH4 pour Linux est assurée directement par ST Microelectronics sous la forme de la distribution STLinux. Dans le cas présent nous avons utilisé les GNU tools de STLinux-2.4 disponibles sur <http://www.stlinux.com/install/getting-started/installing-gnu-tools>.

Il est toujours possible de construire un compilateur spécifique, mais l'utilisation de la version ST assure la *validation* du code généré.

3.1.3 Altera NIOS II

La aussi nous avons choisi d'utiliser les compilateurs fournis par les constructeurs sachant que ceux-ci sont très dépendants des architectures matérielles des FPGA. Ce choix garantit le bon fonctionnement de la génération des binaires.

- La version pour NIOS II sans MMU (GCC3) sur <http://www.niosftp.com/pub/gnutools/nios2gcc-20080203.tar.bz2>
- La version pour NIOS II avec MMU (GCC4) sur <http://uuu.enseirb.fr/~kadionik/nios2-preempt-rt/software/nios2gcc-20090929.tgz>

3.1.4 Xilinx Microblaze

Le compilateur validé par Xilinx est disponible sur <http://xilinx.wikidot.com/mb-gnu-tools>.

3.2. Outil de production Buildroot

Buildroot est un projet très dynamique disponible sur <http://buildroot.uclibc.org>. Il est actuellement utilisé comme système de production de distribution (firmware) pour RTEL4I. Lors de la conception du projet (2008), il n'existait pas de version stable du système Buildroot et il était donc difficilement envisageable d'utiliser un outil disponible uniquement sous SVN.

L'année 2009 a vu une modification majeure de la gestion du projet Buildroot avec l'apparition de versions (release) tous les trois mois, la première étant la 2009.02 (février 2009). Cela permet d'assurer une certaine stabilité du comportement de Buildroot. Actuellement, la version modifiée pour RTEL4I est basée sur Buildroot 2010.11 mais l'architecture modulaire du système permettra d'évoluer plus tard vers la nouvelle version prévue dans quelques jours (2011.05).

4. Test de production du firmware

Dans ce paragraphe, nous allons décrire les méthodes de validation des outils utilisées pour la production de firmware. Ces outils sont intégrés à Buildroot et pilotés par le middleware RTEL4I.

Il existe deux niveaux de validation pour Buildroot :

1. Validation de la compilation (le firmware est produit correctement)
2. Validation du fonctionnement (le firmware fonctionne correctement sur la cible)

4.1. Test de validation de compilation

Ce test est assez facile à automatiser, sachant qu'une erreur de compilation provoque l'échec de l'exécution de la commande `make` utilisée pour dérouler les actions conduisant à la production du firmware.

Dans le cas d'une compilation correcte on a :

```
$ make
/usr/bin/make -j1 silentoldconfig
make[1]: entrant dans le répertoire «
/home/pierre/developpement/OpenWide/buildroot-ow »
...
gzip -9 -c /home/pierre/developpement/OpenWide/buildroot-
ow/output/images/rootfs.tar > /home/pierre/developpement/OpenWide/buildroot-
ow/output/images/rootfs.tar.gz
rm -f /home/pierre/developpement/OpenWide/buildroot-ow/output/build/.fakeroot*
```

On peut tester le résultat par le code de retour de la commande disponible dans la variable automatique `$?` .

```
$ echo $?
0
```

En cas d'erreur, le comportement est différent.

```
$ make
/usr/bin/make -j1 silentoldconfig
make[1]: entrant dans le répertoire «
/home/pierre/developpement/OpenWide/buildroot-ow »

make[1]: quittant le répertoire « /home/pierre/developpement/OpenWide/buildroot-
ow »
tar: Exiting with failure status due to previous errors
make: *** [/home/pierre/developpement/OpenWide/buildroot-
ow/output/build/xenomai-
2.5.5.2/.unpacked] Erreur 2
```

Le code vaut alors 2 (non nul).

```
$ echo $?
2
```

Il est donc aisé d'automatiser le test dans un script qui pourra :

1. Modifier la configuration de Buildroot, soit le fichier `.config` (fichier texte)
2. Lancer la compilation et tester le code de retour.

4.2. Test d'exécution sur QEMU


Le test d'exécution est plus complexe car il ajoute la phase d'installation puis de test réel de la cible. Il est donc nécessaire de :

1. Produire le firmware (test précédent)
2. L'installer sur la cible
3. Rebooter la cible
4. Dérouler une procédure de test liée aux fonctionnalités ajoutées. Par exemple, l'ajout d'un service réseau – comme un serveur HTTP - conduit au test de ce service, soit faire une requête HTTP sur la cible (port TCP 80).

Un émulateur de type QEMU peut grandement faciliter la procédure car les éléments du firmware produits (images noyau et root-filesystem) peuvent être directement utilisées dans QEMU qui est une commande en espace utilisateur, exemple :

```
$ qemu-system-arm -M versatilepb -m 32 -kernel zImage -initrd rootfs.cpio
```

La commande précédente permet de tester le fonctionnement des fichier `zImage` (noyau) et

 open wide INGENIERIE	Validation des outils	Version: 1.0 Auteur: Pierre Ficheux
--	-----------------------	--

rootfs.cpio (root-filesystem) dans l'émulateur QEMU version ARM (qemu-system-arm). L'option -m permet de spécifier la mémoire allouée (ici 32 Mo). On peut donc faire varier cette valeur afin de connaître la quantité de RAM nécessaire pour faire fonctionner la carte avec un firmware donné.

Cette méthode est fréquemment utilisée dans des industries de pointe dans le cas d'un matériel non disponible ou inexistant (en cours de développement). Par contre, cela nécessite de disposer d'un émulateur émulant tous les périphériques nécessaires au test ce qui n'est pas toujours possible sur des architectures moins répandues (FPGA, SH4, ...).

4.3. Test sur cible réelle, utilisation du « virtual lab » (VLAB)

Open Wide a développé depuis plusieurs années (bien avant RTEL4I) un outil permettant de tester des cartes réelles installées sur un laboratoire virtuel connecté au réseau de l'entreprise et accessible par SSH (accès sécurisé à base de clé privé/publique). Le VLAB est un serveur sous Linux disposant de plusieurs éléments physiques :

- Un switch RJ-45 permettant la connexion des cibles
- Un serveur de terminal MOXA (http://www.moxa.com/product/Terminal_Servers.htm) permettant de mettre à disposition un grand nombre de lignes RS-232. Ces lignes sont utilisées pour les consoles des cibles qui le plus souvent en RS-232.
- Une barrette d'alimentation APC (<http://www.apc.com/products/family/index.cfm?id=70>) permettant de piloter à distance (par réseau, protocole TELNET) le redémarrage des cibles

Cet outil permet entre-autres d'effectuer des tests et des développements à distance en cas d'utilisateur délocalisé.

Les deux photos suivantes montrent le VLAB ainsi que certaines cibles du projet.



Figure 1. Console du VLAB, switch et MOXA

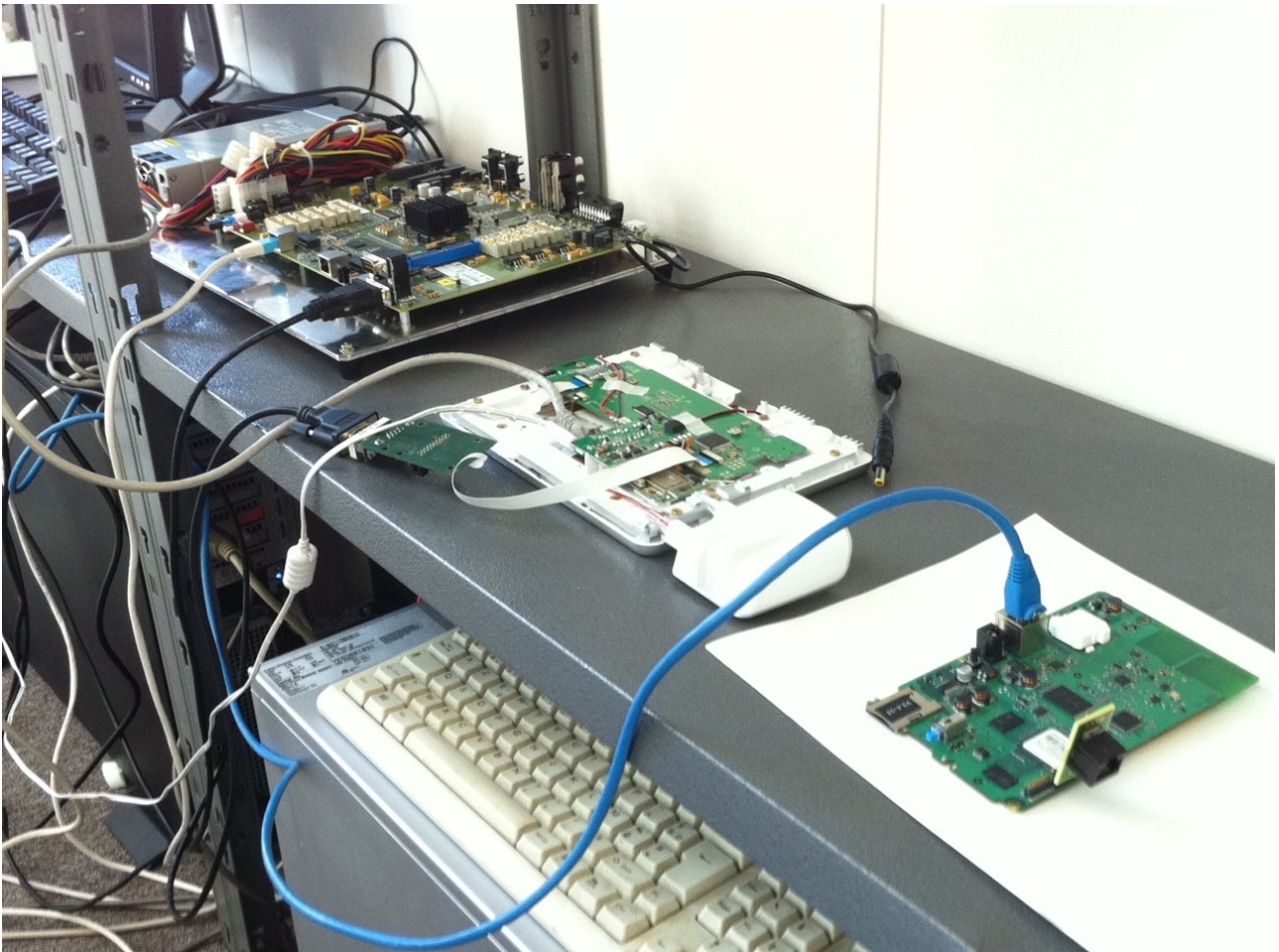


Figure 2. Les cibles SH4, Screen et FemtoCell sur le VLAB

Logiciel du VLAB

Pour utiliser ce matériel, nous avons développé un logiciel écrit en langage Python permettant de gérer le système :

- Affichage des cibles connectées
- Ajout de cibles
- Arrêt d'une cible
- Démarrage d'une cible
- Reboot d'une cible

- Génération des paramètres de la console série

```
$ vlab
/usr/local/bin/vlab:
  <start | stop | restart | list | update>
  <reboot | on | off | status> <board>
  <tty | ip> <board>
```

Liste des cartes :

```
$ vlab list
Board      Power IP           TTY           Speed  Bits Parity Stop
-----
sh4        3      192.168.3.51 /dev/ttyA11  115200  8    N    1
ethylo     23     10.10.10.53  /dev/ttyA14  38400   8    N    1
r570       21     192.168.3.80 /dev/ttyA21  57600   8    N    1
fonera     23     10.10.10.62  /dev/ttyA15  9600    8    N    1
ekinops1   23     192.168.3.50 /dev/ttyA17  38400   8    N    1
brakali    4      192.168.3.229 /dev/ttyA12  115200  8    N    1
v840       2      192.168.3.81 /dev/ttyA22  57600   8    N    1
largo      2      10.10.10.55  /dev/ttyA12  115200  8    N    1
knjn       23     192.168.3.229 /dev/ttyA13  115200  8    N    1
```

Etat de la cible sh4 (OFF)


```
$ vlab status sh4
status 3
OK
3: OFF : Outlet 3
```

Démarrage de la cible *sh4* .

```
pficheux@vlab:~$ vlab on sh4
on 3
OK
3: Outlet 3 : Outlet Turned On
```

Vérification du démarrage de la cible par ping :

```
pficheux@vlab:~$ ping 192.168.3.51
PING 192.168.3.51 (192.168.3.51) 56(84) bytes of data.
64 bytes from 192.168.3.51: icmp_seq=1 ttl=64 time=0.314 ms
64 bytes from 192.168.3.51: icmp_seq=2 ttl=64 time=0.292 ms
64 bytes from 192.168.3.51: icmp_seq=3 ttl=64 time=0.304 ms
```

	Validation des outils	Version: 1.0 Auteur: Pierre Ficheux
---	-----------------------	--

Bien entendu on peut se connecter à la carte par la console série et l'utilitaire minicom :

```
pficheux@vlab:~$ minicom -o -w sh4
```

```
Bienvenue avec minicom 2.1
```

```
OPTIONS: History Buffer, F-key Macros, Search History Buffer, I18n
Compilé le Nov 5 2005, 15:43:56.
```

Tapez CTRL-A Z pour voir l'aide concernant les touches spéciales

```
mb442 login: root
# uname -a
Linux mb442 2.6.32.10_stm24_0201-mb442 #302 PREEMPT Fri May 13 17:02:56 CEST 201
1 sh4 GNU/Linux
```

Test avec le VLAB

Le VLAB permet donc d'effectuer du test automatique à partir des données produites (noyau et root-filesystem). Toutes ces étapes sont intégrables dans un script de test.

- Le noyau Linux de la cible est chargé par le bootloader au démarrage via le protocole TFTP. Il suffit donc de copier le fichier sur le répertoire associé (/tftpboot) sur le VLAB
- Le root-filesystem est monté par NFS-Root (réseau). Le contenu du root-filesystem est extrait à partir de l'archive rootfs.tar.gz produite par Buildroot et placé dans un répertoire du VLAB exporté par NFS (exemple: /home/pficheux/rootfs_test).
- Démarrage d'une session console avec minicom, ce qui permet d'enregistrer les traces sur la console.
- Le reboot de la cible est effectué par la commande `vlab reboot nom_de_cible`.
- On peut détecter le démarrage de la cible par un accès réseau (ping)
- Une fois le démarrage détecté, on peut dérouler les tests spécifiques à la fonction ajoutée.