Réalisation du système pour Olimexino-A13-micro sur SD-card

⚠Les cartes Olimexino-A13-micro (Fig. 1) et Olimexino-A13 sont différentes, ne serait-ce que par la quantité de mémoire disponible. Les informations disponibles sur le web concernant Olimexino-A13 sont donc à transposer aux Olimexino-A13-micro avec précaution.

L'approche proposée n'exploite pas de paquet binaire pré-compilé mais vise, par soucis de cohérence des outils, à fabriquer soi-même tous ses outils à partir des sources, avec l'aide de `buildroot`. Cet outil propose un environnement de développement **cohérent** qui fournit *bootloader*, *toolchain*, noyau et système de fichiers *rootfs*.

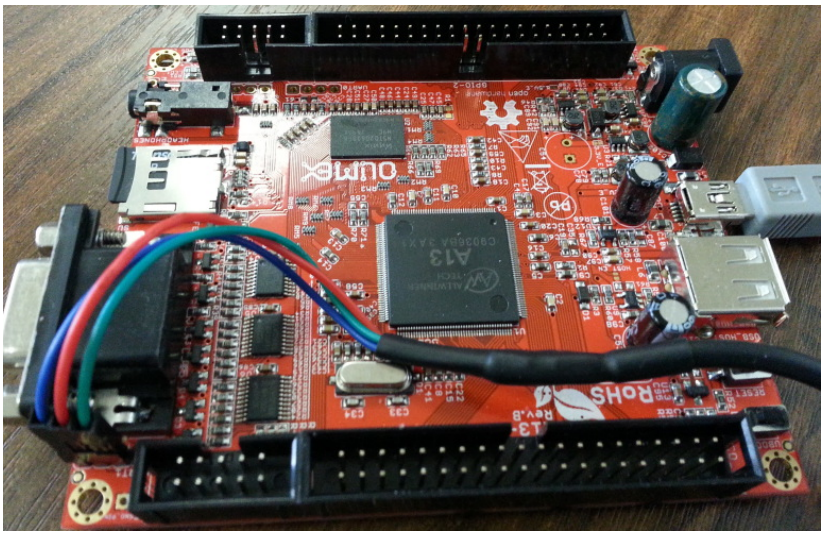


FIGURE 1 – Carte Olinuxino A13 micro équipée de son cable de communication RS232-USB. Noter l'ordre des couleurs des fils de liaison. Dans ce contexte, le câble USB gris ne sert qu'à alimenter la carte Olinuxino A13 micro.

1 Pré-requis

```
# apt-get install build-essential libncurses5-dev u-boot-tools
# apt-get install git libusb-1.0-0-dev pkg-config
```

Les commandes précédées d'un $ doivent être exécutées en mode utilisateur.
Les commandes précédées d'un # doivent être exécutées en mode super-utilisateur.

2 Génération de tous les outils depuis les sources : `buildroot`

Buildroot [1] est un outil intégré permettant de

1. compiler sa toolchain pour la cible appropriée (dans `output/host/usr/bin/`),
2. compiler son noyau Linux,
3. compiler ses outils GNU.

L'avantage de buildroot [1], comme de open-embedded [2], est de fournir un environnement intégré et cohérent qui cache un

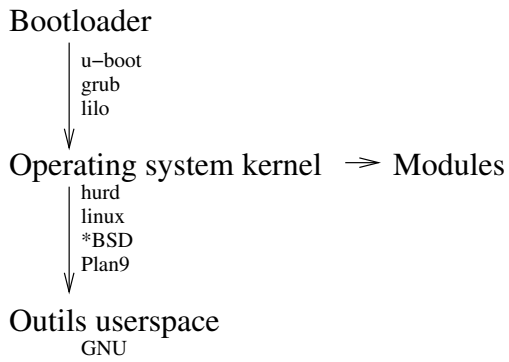


FIGURE 2: Éléments nécessaires au fonctionnent d'un système d'exploitation : les outils générant toutes ces étapes doivent être cohérents.

certain nombre de détails au développeur (Fig. 2). L'inconvénient de buildroot, comme de open-embedded, est de fournir un environnement intégré et cohérent qui cache un certain nombre de détails au développeur (!). Nous allons profiter de cet environnement de travail pour compiler tous les outils nécessaires au travail sur A13-Olinuxino-micro sans dépendre d'une distribution binaire externe, et en particulier pour ajouter l'extension temps-rélle de Linux nommée Xenomai [2]. Cette extension nécessite, en plus d'une mise à jour du noyau, des outils en espace utilisateur que buildroot peut gérer. Le port de Xenomai à Olimexino-A13 a été réalisé par Paul Crubillé (Université Technologique de Compiègne) [3].

Buildroot, disponible à `https://github.com/buildroot/buildroot`, a été adapté à l'Olimex A13-micro au travers de patchs à appliquer d'une part à `buildroot` (configuration des ressources mises à disposition par la plateforme matérielle), et d'autre part aux sources du noyau Linux téléchargées lors de la compilation de `buildroot`. Ces patchs sont en cours d'acceptation par la communauté de développeurs de logiciels libres (noyau Linux et buildroot) afin de garantir la pérénité des solutions proposées et leur évolution avec les environnements dans lesquels ils s'appliquent.

3 Compiler une image Linux sans support Xenomai (sans support temps réel)

Télécharger (`git clone`) le contenu du dépôt de `https://github.com/trabucayre/buildroot` : cette archive contient une version officielle de `buildroot` complétée d'un certain nombre de correctifs pour la carte qui nous intéresse. La configuration de l'environnement de travail pour appliquer les particularités à la carte A13-micro s'obtient par la commande

```
make olimex_a13_olinuxino_micro_defconfig
```

Ensuite, `make` génère une première image fonctionnelle qui pourra être transférée sur la carte SD selon la procédure décrite dans `board/olimex/a13_olinuxino/readme.txt` (en pratique, exécuter le script `board/olimex/a13_olinuxino/make-sdimg.sh` avec les arguments de l'emplacement des images, `output/images`, et l'emplacement de la carte SD cible, la plupart du temps `/dev/sdb` ou `/dev/mmcblk0`.

⚠**Attention** : le contenu de la carte SD, ou de tout support pointé par le dernier argument de cette commande, sera **irrémédiablement** perdu. Vérifier à deux fois le nom du périphérique cible de l'image issue de buildroot.

3.1 Ethernet sur USB Gadget

Nous avons activé, par défaut, le support de l'émulation d'une liaison TCP/IP sur support physique USB, tel que fourni par le module `gadget ethernet` de USB (aussi connu sous le nom du fichier géné, `g_ether`).

Ce résultat aurait pu s'obtenir manuellement par `make linux-menuconfig` depuis la racine de l'arborescence buildroot, puis en suivant la procédure proposée à `http://linux-sunxi.org/USB_Gadget`, mais pour un noyau relativement ancien et selon une procédure qui diffère quelque peu pour le noyau qui nous concerne. La configuration matérielle décrite dans le *devicetree* et décrivant l'interface USB-OTG a été acceptée dans le noyau officiel Linux et ne nécessite pas d'intervention manuelle du développeur.

Si l'émulation de la liaison USB est fonctionnelle (sur la carte A13-micro connectée par la liaison RS232 [4], `modprobe g_ether` suivi de `ifconfig usb0` pour voir si le périphérique existe), il reste à configurer le réseau. On vérifiera néanmoins que le PC est bien équipé des modules nécessaires à l'émulation d'ethernet sur USB : `dmesg` doit indiquer

```
usb 3-1: new full-speed USB device number 14 using xhci_hcd
usb 3-1: New USB device found, idVendor=0525, idProduct=a4a2
usb 3-1: New USB device strings: Mfr=1, Product=2, SerialNumber=0
usb 3-1: Product: RNDIS/Ethernet Gadget
usb 3-1: Manufacturer: Linux 3.4.90 with sw_usb_udc
```

1. `buildroot.uclibc.org/`
2. `www.openembedded.org`
3. `http://permalink.gmane.org/gmane.linux.real-time.xenomai.users/16199`
4. lancer `minicom` après avoir branché le cable RS232-USB, en 115200 bauds, 8N1 : un prompt demandant login/mot de passe doit apparaître à l'appui de la touche ENTER

```
cdc_subset: probe of 3-1:1.0 failed with error -22
cdc_ether 3-1:1.0 usb0: register 'cdc_ether' at usb-0000:00:14.0-1, CDC Ethernet Device, 06:28:1e:fd:eb
```

La façon courte d'accéder au réseau – qui doit être reconfiguré à chaque redémarrage au travers de l'interface RS232, est `ifconfig usb0 172.16.1.2` sur l'Olinuxino A13 et `ifconfig usb0 172.16.1.1` sur le PC. Noter que pour pouvoir initier une connection `ssh`, le compte `root` doit être muni d'un mot de passe sur la carte Olinuxino (`passwd root` par exemple). Une version plus pérenne de la configuration du réseau est décrite ci-dessous.

3.2 Configuration du réseau

3.2.1 Coté cible

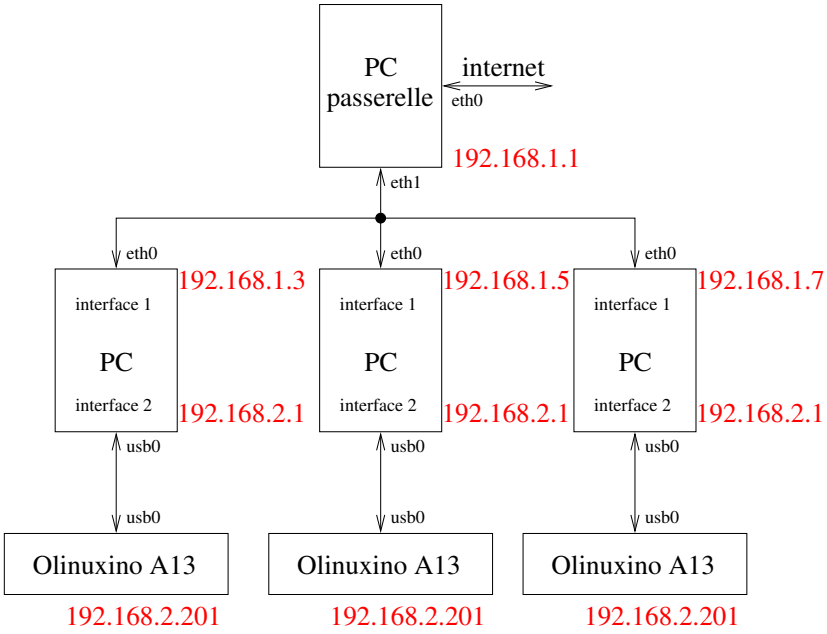


FIGURE 3: Architecture du réseau TCP/IP

Après s'être connecté sur la cible Olinuxino, le serveur de nom de domaine (qui effectue la traduction entre nom de domaine et adresse IP) – DNS – est informé dans le fichier `/etc/resolv.conf`. Cette information n'est nécessaire que pour accéder à un domaine sur internet (par exemple `free.fr` ou `google.com`) et est inutile tant que nous nous limitons à une connexion locale où l'interlocuteur est renseigné par son adresse IP.

Le format des informations dans `/etc/resolv.conf` est
`nameserver 194.57.91.200`
`nameserver 194.57.91.201`
avec 194.47.91.20{0,1} les serveurs de nom de domaine de l'université de Franche Comté.

Nous pouvons tenter d'automatiser la configuration sur une adresse IP statique de l'interface `usb0` en informant le fichier `etc/network/interfaces` de la façon suivante :

```
auto usb0
iface usb0 inet static
address 192.168.2.200
netmask 255.255.255.0
gateway 192.168.2.1
```

Cette configuration automatise la séquence de commandes `ifconfig` et `route` vues auparavant : ici, le routeur par défaut (pour gérer les paquets qui ne sont pas destinés à un interlocuteur local) est d'adresse IP 192.168.2.1.

L'interface réseau `usb0` peut être configuée automatiquement, même si la fonctionnalité n'est fournie que par un module noyau chargé dynamiquement, en ajoutant au fichier `etc/modules` :

```
g_ether
```

Cependant, buildroot ne supporte pas par défaut le chargement automatique de modules noyau (comme le fait une distribution de type Debian). Il faut donc ajouter un script de chargement automatique de module noyau dans le répertoire contenant les scripts exécutés au démarrage, `/etc/init.d`. Un exemple d'un tel script inspiré de Debian est

```
1  #!/bin/sh -e
2
3  # Silently exit if the kernel does not support modules.
4  [ -f /proc/modules ] || exit 0
5  [ -x /sbin/modprobe ] || exit 0
6
7  PATH='/sbin:/bin'
8
9  load_module() {
10   local module args
11   module="$1"
```

```
12   args="$2"
13
14   echo "Loading kernel module $module"
15   modprobe $module $args || true
16 }
17
18 files="/etc/modules"
19 if [ "$files" ] ; then
20   grep -h '^[^#]' $files |
21   while read module args; do
22     [ "$module" ] || continue
23     load_module "$module" "$args"
24   done
25 fi
```

que nous stockons dans `/etc/inid.d/S02kmod`. Il reste cependant un problème de chargement dynamique du module `g_ether` à cause d'un dysfonctionnement du passage de l'USB OTG en *device* ... pour le moment nous devons nous contenter de charger `g_ether` manuellement.

3.2.2 Coté hôte (PC)

Interface réseau

Ajouter au fichier `/etc/network/interfaces` la commande de configuration par un IP statique de l'interface `usb0` :

```
auto usb0
allow-hotplug usb0
iface usb0 inet static
address 192.168.2.1
netmask 255.255.255.0
```

NFS (Network File System)

Le partage de répertoire par NFS permet de faire apparaître une partie du disque dur du PC comme un répertoire de la carte embarquée Olimexino. Ce partage virtuel de fichiers au travers d'une connexion réseau fournit un environnement souple de développement puisqu'à l'issue de la cross-compilation d'un binaire sur le PC, l'exécutable résultant est immédiatement disponible dans le répertoire partagé pour exécution sur la plateforme embarquée.

Créer pour ce fairem sur le PC, un répertoire `/home/utilisateur/target` qui sera le répertoire partagé, et dans le fichier `/etc/exports` de l'hôte, indiquer que nous voulons autoriser le partage de ce répertoire par l'ajout de la ligne : `/home/utilisateur/target *(rw,sync)`

Relancez nfs :

```
#/etc/init.d/nfs-kernel-server restart
```

Ce répertoire permettra d'échanger des fichiers entre l'hôte et la cible, par exemple les exécutables que nous aurons cross-compilés sur le PC et que nous désirons exécuter sur l'Olinuxino A13 micro. Sur la carte A13, le répertoire est monté par `mount -o nolock IP_PC:/home/utilisateur/target /mnt` : le contenu de `/home/utilisateur/target` de l'hôte apparaît désormais dans `/mnt` de la cible.

Connexion à internet en configurant le PC comme passerelle

Nous ne développons pas ici la possibilité d'utiliser le PC comme une passerelle qui se charge de la traduction d'adresses IP afin de permettre à la carte Olinuxino A13 de se connecter à internet. L'outil qui permet de faire cette traduction se nomme `iptables` : un exemple de configuration qui transfère tous les paquets entre les interfaces eth0 et usb0 est

```
monpath=/sbin

echo "1" > /proc/sys/net/ipv4/ip_forward
echo "1" > /proc/sys/net/ipv4/ip_dynaddr
${monpath}/iptables -P INPUT ACCEPT
${monpath}/iptables -F INPUT
${monpath}/iptables -P OUTPUT ACCEPT
${monpath}/iptables -F OUTPUT
# ${monpath}/iptables -P FORWARD DROP
${monpath}/iptables -P FORWARD ACCEPT
```

```
${monpath}/iptables -F FORWARD
${monpath}/iptables -t nat -F

${monpath}/iptables -A FORWARD -i eth0 -o usb0 -m state --state ESTABLISHED,RELATED -j ACCEPT
${monpath}/iptables -A FORWARD -i usb0 -o eth0 -j ACCEPT
${monpath}/iptables -A FORWARD -j LOG

${monpath}/iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

3.3 Serveur web

L'image de buildroot pour Olinuxino A13-micro contient un serveur web léger, qui supporte les scripts CGI, nommé `boa`. Pour s'en servir :

1. `mkdir -p /usr/local/boa`
2. `chown nobody.nogroup /usr/local/boa`
3. `chmod 755 /usr/local/`
4. `chmod 755 /usr/local/boa` (vérifier que l'utilisateur nobody.nogroup peut accéder à ce répertoire)
5. placer dans `/usr/local/boa/boa.conf` un fichier de configuration contenant par exemple
   ```
   Port 80
   User nobody
   Group nogroup
   ErrorLog /usr/local/boa/error_log
   AccessLog /usr/local/boa/access_log
   DocumentRoot /usr/local/boa/
   UserDir public_html
   DirectoryIndex index.html
   DirectoryMaker /usr/lib/boa/boa_indexer
   # DirectoryCache /var/spool/boa/dircache
   KeepAliveMax 1000
   KeepAliveTimeout 10
   MimeTypes /etc/mime.types
   DefaultType text/plain
   # Uncomment the next line if you want .cgi files to execute from anywhere
   AddType application/x-httpd-cgi cgi
   ScriptAlias /cgi-bin/ /usr/local/boa/cgi/
   ```
   ⚠Ne jamais lancer de service avec les autorisations d'administrateur (`root`). Même si une telle configuration rend l'accès aux ressources matérielles plus facile, elle met en danger le principe de séparation des autorisations d'unix selon les services (exemples de mauvaises configurations de serveurs web : *Exploiting Network Surveillance Cameras Like a Hollywood Hacker*, Black Hat 2013, à `https://www.youtube.com/watch?v=B8DjTcANBx0`). Ici, l'utilisateur `nobody` et le groupe `nogroup` n'ont aucun pouvoir sur le système et une défaillance de sécurité du serveur web ne se traduit pas par une remise en cause de l'intégrité du système.
6. lancer `boa` par `boa -f /usr/local/boa/boa.conf`
7. connecter un client web vers l'adresse IP de la carte Olinuxino A13-micro : le fichier `index.html` que nous aurons placé dans `/usr/local/boa` y sera afficheé si les permissions appropriées lui sont accordées.

Pour exécuter un script cgi, nous pouvons créer le sous répertoire `/usr/local/boa/cgi` (en accord avec la dernière ligne du fichier de configuration) et y placer par exemple un script shell

```
#!/bin/sh
echo -e "Content-type: text/html\r\n\r\n"
echo "Hello CGI"
```

qui sera appelé en faisant pointer le client web vers `IP/cgi/script.cgi`. On pensera naturellement à autoriser l'exécution (`chmod 755`) du script shell. Cette méthode de travail permet donc de générer dynamiquement des pages web, par exemple pour représenter la mesure de capteurs.

3.4 Activation de la sortie VGA

Afficher une console sur un écran connecté au port VGA (DB-15 femelle) nécessite de valider 3 étapes :

1. décrire la matériel nécessaire à l'accès aux broches commandant le périphérique. On notera qu'il s'agit des mêmes broches pour le port LCD et VGA, avec dans le second cas de la sortie analogique une conversion numérique-analogique par sommation des courants traversant des résistances pour commander les intensités des voies vertes, rouges et bleues du VGA,
2. charger le module noyau commandant ces broches, et en particulier fournissant l'interface *framebuffer* pour l'affichage depuis l'espace utilisateur,
3. indiquer qu'une console doit être affichée sur le port VGA (configuration du système en espace utilisateur).

Pour le premier point, le support matériel a été pris en charge dans la configuration du *devicetree*.

Pour le second point, `uboot` semble prendre en charge toutes les configurations nécessaires à l'activation de la sortie VGA. Sur les noyaux Linux de la série 3.xx, nous devions activer dans la configuration du noyau (accessible par `make linux-menuconfig` dans le répertoire de buildroot) `DISP Driver Support(sunxi)` et `LCD Driver Support(sunxi)` sous `Device Drivers → Graphics support → Support for frame buffer devices`. On pourra alternativement directement modifier le fichier `.config` du noyau en activant `CONFIG_FB_SUNXI=y` et `CONFIG_FB_SUNXI_LCD=y`.

Enfin, le troisième point (Fig. 4) s'obtient en ajoutant dans `/etc/inittab` la ligne
`tty1::respawn:/sbin/getty -L tty1 115200 vt100`

⚠Pour les noyaux exploitant `script.fex` (ce qui *exclut* donc les noyaux de version 4.x), en cas de Kernel Panic lors du démarrage du noyau (support VGA en statique) ou lors du chargement du module `lcd.ko` (support VGA sous forme de module dynamique), l'erreur provient de l'absence de renseignement du support VGA dans le script de configuration matériel `script.bin` chargé dans la première partition de la carte SD. Afin de pallier à cette déficience, on pourra manuellement remplacer le fichier par défaut par celui généré manuellement en tenant compte de `a13/a13-olinuxinom.fex`. Pour ce faire

1. `export PATH=$PATH:/home/mon_chemin/buildroot-a13-olinuxino/output/host/usr/bin` ajoute le chemin contenant les binaires sur la plateforme hote (PC), et en particulier `fex2bin` nécessaire à convertir le fichier lisible par un humain en format compréhensible par le système

2. `fex2bin sunxi-boards/sys_config/a13/a13-olinuxinom.fex output/images/script.bin` tel que obtenu auprès de
   `https://github.com/linux-sunxi/sunxi-boards/blob/master/sys_config/a13/a13-olinuxinom.fex`

3. copier `output/images/script.bin` dans la première partition de la carte SD

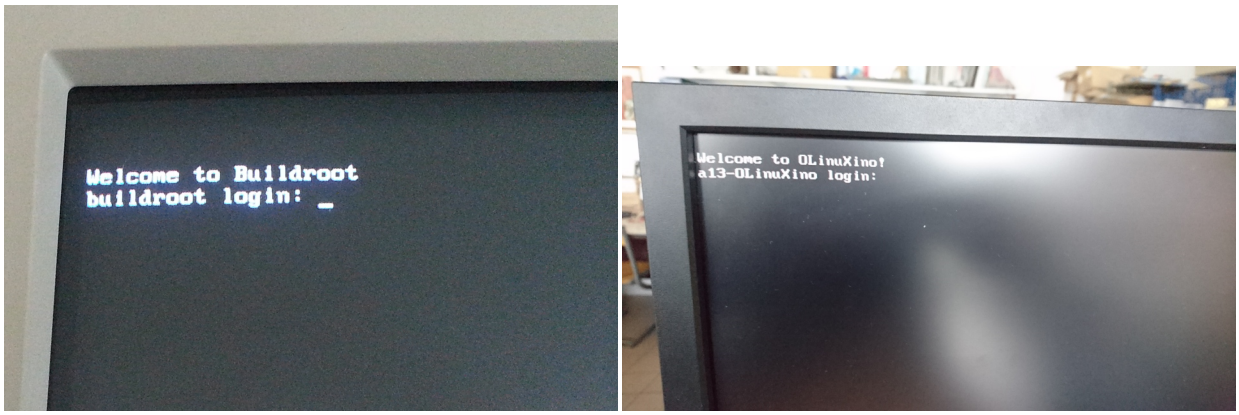


FIGURE 4 – Login au lancement de la console après démarrage et activation de la sortie VGA (ancien et nouveau noyau).

3.5 Interface graphique programmée en Qt5

Qt5 est un ensemble de bibliothèques programmées en C++ permettant de générer du code compatible multiplateformes, que ce soit entre systèmes d'exploitation (MS-Windows, X11 notamment sur GNU/Linux, OS X ou iOS chez Apple) ou pla-

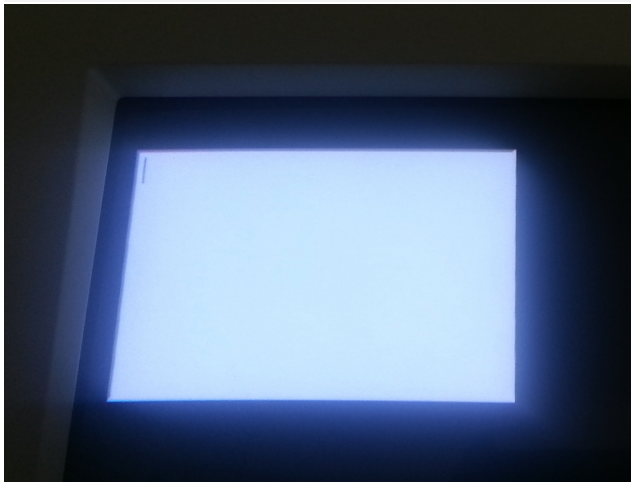
teformes matérielles (x86, MIPS, ARM, SPARC ...). Malgré sa lourdeur, inhérente à toute interface graphique, elle permet de mettre en valeur les performances du système d'exploitation exécuté sur système embarqué.

La compilation de Qt5 par buildroot nécessite le support C++ – qui n'est pas actif par défaut. Recompiler la *toolchain* avec support C++ est garanti en effaçant le répetoire `output` (`rm -rf output`) de buildroot et en re-lancçant `make`.

Une erreur de compilation dans les modules de communication réseau (`network`) de Qt5 se corrige par la modification décrite à `https://forum.qt.io/topic/40015/qdnslookup_unix-cross-compilation-error/3`. Mis à part ce désagrément, la compilation se finit convenablement avec l'obtention de `qt5bsae`, `network module`, `gui module`, `widgets module`, `linuxfb support` et `directfb support`.

Une fois le système installé, il nous faut développer une application de test. Le programme trivial proposé (pour Qt4.8) à `http://doc.qt.io/qt-4.8/gettingstartedqt.html` est sauvé dans un fichier d'extension `.cpp` (Fig. 5).

```cpp
#include <QApplication>
#include <QTextEdit>

int main(int argv, char **args)
{
 QApplication app(argv, args);
 QTextEdit textEdit;
 textEdit.show();
 return app.exec();
}
```



```cpp
// http://doc.qt.io/qt-4.8/gettingstartedqt.html

#include <QApplication>
#include <QTextEdit>
#include <QPushButton>
#include <QVBoxLayout>

int main(int argv, char **args)
{
 QApplication app(argv, args);

 QTextEdit *textEdit = new QTextEdit;

 QPushButton *quitButton = new QPushButton("&Quit");
 QObject::connect(quitButton, SIGNAL(clicked()), qApp,→
     ↪ SLOT(quit()));

 QVBoxLayout *layout = new QVBoxLayout;
 layout->addWidget(textEdit);
 layout->addWidget(quitButton);

 QWidget window;
 window.setLayout(layout);
 window.show();

 return app.exec();
}
```

FIGURE 5 – Gauche : exemple d'éditeur de texte, objet fourni par la bibliothèque Qt. Droite : exemple à peine plus évolué avec un bouton.

⚠L'extension du fichier est importante : une extension `.c` se solde par l'appel à `gcc` qui ne comprend pas la syntaxe du C++ et ne sait pas appeler les bibliothèques appropriées.

Qt propose un mécanisme de compilation qui génère le `Makefile` à partir d'un fichier de configuration d'extension `.pro`. Ce fichier est généré automatiquement par `qmake -project`. Cependant, nous devons prendre soin d'appeler `qmake` depuis le répertoire `output/host/usr/bin` de buildroot *et non le qmake potentiellement installé sur l'hôte (PC).* Ayant généré le fichier de configuration `.pro`, nous y ajoutons les modules de Qt appelés par notre programme, dans notre cas le module widgets, par

```
QT += widgets
```

pour finalement obtenir un fichier de configuration de la forme

```
TEMPLATE = app
TARGET = Qt
INCLUDEPATH += .

QT += widgets

# Input
SOURCES += 2nd_test.cpp
```

Nous appelons à nouveau `BR/output/host/usr/bin/qmake` – cette fois sans l'option `-project` – pour générer le Makefile, et nous convertissons le code source en binaire exécutable par `make`.

Une fois la compilation achevée, deux résultats sont obtenus :

— l'image (rootfs) GNU/Linux à destination de la cible ARM comporte les bibliothèques Qt
— l'hôte (le PC) est muni des outils Qt de configuration et de compilation des programmes écrits en C++ selon une syntaxe faisant appel aux bibliothèques Qt.

Le programme s'exécute depuis la plateforme Olinuxino A13-micro en précisant que la sortie graphique se trouve sur le *framebuffer* : après transfert du fichier sur la carte SD de l'Olinuxino, nous exécutons `Qt -platform linuxfb` et, si tout se passe bien, une fenêtre d'éditeur de texte s'affiche dans le coin en haut à gauche de l'écran. Les autres plateformes supportées sont par exemple `Qt -platform directfb` (Fig. 5)

Une fois la méthode de compilation comprise, la langage Qt lui-même est documenté et des exemples fournis dans l'archive à `BR/output/build/qt5base-5.3.1/examples`. Nous avons ainsi validé le bon fonctionnement de l'exemple `digiflip` ou `styleexample` (Fig. 6).

FIGURE 6 – Gauche : l'exemple `analogclock`. Milieu : l'exemple `digiflip`. Droite : l'exemple `styleexample`. Tous trois sont exécutés sur la plateforme Olinuxino A13-micro, sortie VGA en framebuffer `linuxfb`.

4 Compiler une image Linux avec support Xenomai (extension-temps réel)

Le buildroot issu de `https://github.com/trabucayre/buildroot-a13-olinuxino` – proposé par G. Goavec-Mérou selon la procédure décrite en annexe A.1 – se configure pour la cible Olimex-A13micro avec support Xenomai au moyen de

```
make a13_olinuxino_micro_xenomai_defconfig
```

qui définit un certain nombre de paramètres par défaut qui nous seront utiles. En particulier, cette configuration amène les modifications nécessaires aux sources d'un noyau Linux un peu ancien (3.4.24) pour supporter la couche ipipe faisant le lien entre le matériel et le noyau, et ainsi permettre le support de Xenomai.

Afin d'ajuster `xenomai` à ce nouvel environnement de travail

1. vérifier que les options temps réel sont active, et en particulier les applications associées (outils de test – *testsuite* – et `rt-tests`). Pour ce faire, `Target packages` → `Real-Time` → `Xenomai Userspace` → `Install testsuite`. Par ailleurs, sélectionner la version de Xenomai manuellemment en remplissant le champ `Custom Xenomai version` par `2.6.3`,
2. il peut être judicieux d'installer quelques fonctionnalités additionnelles accessibles depuis le shell, par exemple `screen` dans `Target packages` → `Shell and utilities` → `screen` ou `Target packages` → `System tools` → `cpuload`. `uudecode`, qui nous sera utile pour transférer des fichiers avec le PC, est supporté par défaut par `busybox` (outil qui encapsule en un seul binaire toutes les fonctions habituellement fournies par le shell).
3. vérifier la configuration du *bootloader* en précisant que nous avons affaire à un A13-Olinuxino-*micro*. Pour ce faire, `Bootloaders` → `U-Boot board name` → `A13-OLinuXinoM` (noter le `M` à la fin du nom de la configuration – cette correction se fait dans le menu `U-Boot Sunxi` et *non* `U-Boot` qui est désactivé).

En cas de problème de type `Legacy config options`, éditer `configs/a13_olinuxino_defconfig` et désactiver `INIT_TOOLS` (*i.e.* retirer `BR2_PACKAGE_MODULE_INIT_TOOLS=y`), activer `KMOD` (`BR2_PACKAGE_KMOD=y` et `BR2_PACKAGE_KMOD_TOOLS=y` avant l'option concernant `UEMACS` – ces opérations peuvent aussi se faire dans `Target Packages` → `System Tools` et activer `kmod` et `kmod utilities`). Comme nous utilisons un noyau fourni sous forme d'archive `.tar.gz`, nous n'avons pas besoin de la version git du noyau et pouvons nous contenter de vider la ligne contenant cette url dans `Legacy config options`.

Ces opérations s'effectuent suite à `make menuconfig` avant de finalement se conclure par `make`. La compilation à proprement parler prend environ 35 minutes, en fonction de la bande passante du réseau et de la puissance de l'ordinateur. En cas d'erreur concernant `Inconsistent kallsyms data`, relancer comme indiqué `make KALLSYMS_EXTRA_PASS=1`.

Une fois la compilation achevée et la carte SD flashée (⚠ *attention* au nom du périphérique contenant la carte SD – risque de destruction du disque dur si le nom de ce périphérique est erroné!) selon les instructions de buildroot, GNU/Linux est exécuté sur le circuit A13-Olinuxino-micro : le seul compte est `root`, sans mot de passe.

Les latences de l'option temps-réel [3] se qualifient par `echo "0" > /proc/xenomai/latency` suivi de `latency -p 100`. On observera la variation des latences quand, dans un second terminal, le processeur est chargé par `while ( true ) ; do echo toto;done`. Cette dernière commande se lance dans un second shell accessible par `screen` (CTRL-A c pour créer une nouvelle session du shell, CTRL-A a pour changer de session).

```
# latency -p 100
== Sampling period: 100 us
== Test mode: periodic user-mode task
== All results in microseconds
warming up...
RTT|  00:00:01  (periodic user-mode task, 100 us period, priority 99)
RTH|----lat min|----lat avg|----lat max|-overrun|---msw|---lat best|--lat worst
RTD|      5.083|      5.166|     11.583|       0|     0|      5.083|     11.583
RTD|      5.749|      5.874|     14.124|       0|     0|      5.083|     14.124
RTD|      5.708|      5.874|     11.083|       0|     0|      5.083|     14.124
RTD|      5.708|      5.874|     10.999|       0|     0|      5.083|     14.124
RTD|      5.749|     10.374|     18.999|       0|     0|      5.083|     18.999  <- lancement dans un
RTD|      6.416|     10.624|     18.624|       0|     0|      5.083|     18.999  <- second terminal de
RTD|      6.291|     10.624|     18.291|       0|     0|      5.083|     18.999  <-  while ( true ) ; do
RTD|      6.124|     10.666|     18.249|       0|     0|      5.083|     18.999  <-    echo toto;done
RTD|      5.666|      7.499|     17.458|       0|     0|      5.083|     18.999
RTD|      5.708|      5.874|     12.124|       0|     0|      5.083|     18.999
RTD|      5.749|      5.874|     11.291|       0|     0|      5.083|     18.999
```

5 Transfert de fichiers sur la liaison asynchrone série

Dans les sections qui suivent nous compilerons nos propres exécutables sur PC à destinatin du processeur ARM équipant Olinuxino-A13micro. Comment transférer les fichiers du PC sur la plateforme embarquée? La solution de l'émulation d'une interface sur USB (section 3.1) n'est pas viable car ce pilote ne semble pas fonctionnel dans la version 3.24 du noyau Linux.

La liaison RS232 (Fig. 1) sert déjà deux fonctions : de terminal (lancé par `/etc/inittab` pour lier `getty` au port série) et de console lancée par le noyau au démarrage. Nous voulons en plus nous servir de ce port pour transférer des fichiers. Nous pourrions évidemment, depuis le PC, `cat binaire > /dev/ttyUSB0` pour envoyer le fichier `binaire` vers le port série, mais ceci se solde par une déconnexion sur la système embarqué car un certain nombre de caractères binaires induisent une réaction non-voulue du terminal. Nous devons donc nous garantir que seuls des caractères ASCII sont transmis : nous utiliserons pour cela un outil classique d'encodage des fichiers binaires dans les couriers électroniques, `uuencode`.

Sur le PC : `cat binaire | uuencode mon_executable > /dev/ttyUSB0` pour envoyer une conversion ASCII du programme `binaire` qui sera réceptionné dans un fichier nommé, après décodage, `mon_executable`.

Sur Olinuxino : `cat < /dev/ttyS0 > mon_executable.ascii` réceptionne le fichier (en fin de transmission, CTRL-D pour arrêter la liaison) et le fichier ASCII est converti en binaire par `uudecode mon_executable.acsii # genere mon_executable`. Ce binaire est rendu exécutable par `chomd 755 mon_executable` puis lancé par `./mon_executable`. En cas d'insulte du type `No such file or directory`, des bibliothèques chargées dynamiquement manquent : il faut, sur la carte SD (rootfs supposé sur la seconde partition) : `cp -i /usr/arm-linux-gnueabihf/lib/* /mnt/sdb2/lib/` en refusant d'écraser les fichiers déjà présents. En cas de `Permission denied`, penser à donner les droits d'exécution au fichier (`chmod 755 binaire`).

6 Cross compiler : clignotement d'une LED

Cross-compiler un programme pour Olinuxino-A13 depuis un PC revient à utiliser le compilateur approprié pour générer un binaire compréhensible par un processeur d'architecture ARM. Afin de faire clignoter une diode, nous sommes informés à `http://olimex.wordpress.com/2012/10/23/a13-olinuxino-playing-with-gpios/` que la diode verte de statut se

trouve sur le port GPIO_G9. Une bibliothèque en espace utilisateur accédant directement aux adresses des ports (sans passer par le noyau Linux) est mise à disposition à `http://docs.cubieboard.org/tutorials/common/gpio_on_lubuntu` : nous cross-compilons un exemple trivial d'allumage et d'extinction de diode nommé `example.c`

```
#include <stdlib.h>
#include <stdio.h>
#include "gpio_lib.h"
#define PG9     SUNXI_GPG(9)

int main()
{int i,status=0;

 if(sunxi_gpio_init())                {printf("Failed init  \n");return -1;}
 if(sunxi_gpio_set_cfgpin(PG9,OUTPUT)) {printf("Failed config\n");return -1;}

 while (1) {
    sunxi_gpio_output(PG9,status); sleep(1);
    status^=0xff;
 }
 sunxi_gpio_cleanup();
 return 0;
}
```

par

```
arm-linux-gnueabihf-gcc -c example.c
arm-linux-gnueabihf-gcc -c gpio_lib.c
arm-linux-gnueabihf-gcc -o example example.o gpio_lib.o
```

Envoyer le fichier binaire depuis le PC vers la carte Olinuxino et constater le résultat. Reproduire en encodant par `uuencode`.

En cas d'absence de certaines bibliothèques (`Command not found`), ajouter dans le répertoire `lib` de la seconde partition de la carte SD contenant le système d'exploitation les fichiers `libc.so.6` et `ld-linux.so.3` (cf le résultat de `ldd example`).

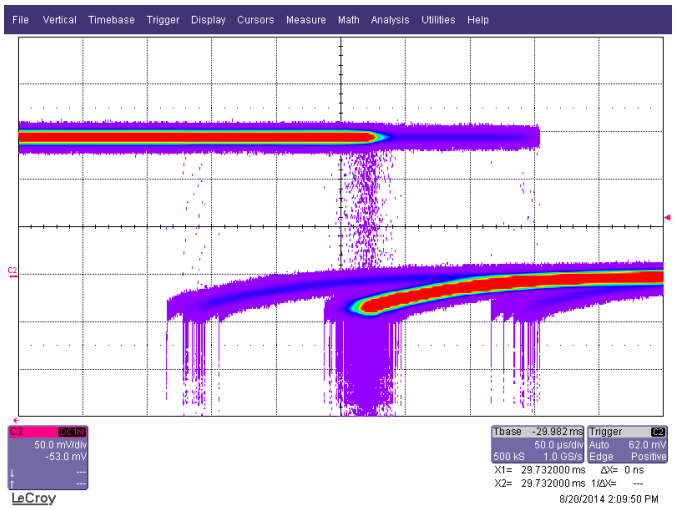
Ayant validé la capacité à commander l'allumage et l'extinction d'une diode depuis l'espace utilisateur, nous allons exploiter cette fonctionnalité pour comparer les latences lors de l'exécution d'une tâche périodique avec ou sans l'option temps-réel. Les quatre options qui s'offrent à nous, excluant la solution d'un module noyau, sont des temporisations par `sleep` ou par *timer*, avec ou sans l'extension temps-réel de Xenomai. Dans tous les cas, nous lançons sur l'Olinuxino-A13-micro, auquel nous nous connectons par le port série virtuel au moyen de `kermit -c`, une session `screen`. Un terminal sert à lancer le programme commutant l'état de la diode de statut (verte) tandis que le second terminal charge ou non le processeur en exécutant `while (true); do echo toto;done`.

7 Qualification des latences

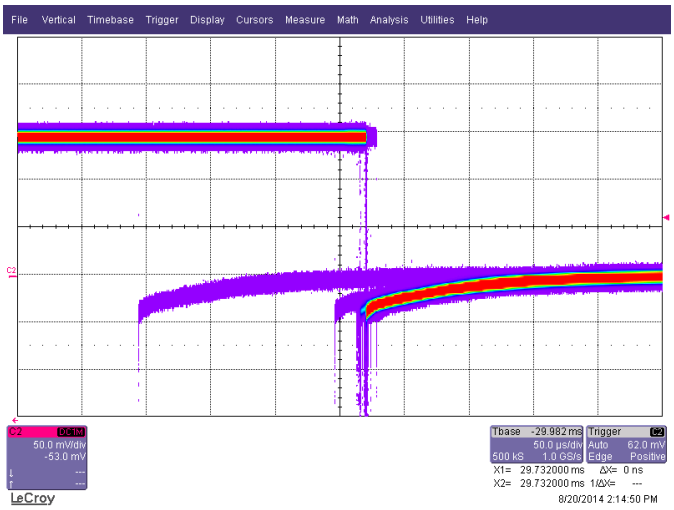
Les quatre cas ci-dessous considèrent les attentes entre deux commutations de la LED verte soit par fonction `sleep` (délai prédéfini), soit par un appel à une fonction par *timer*. Hors Xenomai, l'implémentation de cette méthode par signaux donne un résultat catastrophique dès que le processeur est chargé. Dans les autres cas, la qualification des latences se fait en déclenchant un oscilloscope numérique sur le front montant des crénaux et en observant la date de chute du crénau. Si les attentes respectaient parfaitement le chronogramme prévu, les fronts descendants devraient se superposer. L'intervalle de temps dans lequel ces fronts sont observés permet de qualifier la résistance à la charge et le respect de latences maximum dans les diverses conditions analysées. Le cas des modules noyau n'est pas abordé.

7.1 Attente par `sleep`, sans Xenomai

Le programme reprend l'exemple vu plus haut de commutation des diodes, avec une attente de 500 ms.



En charge

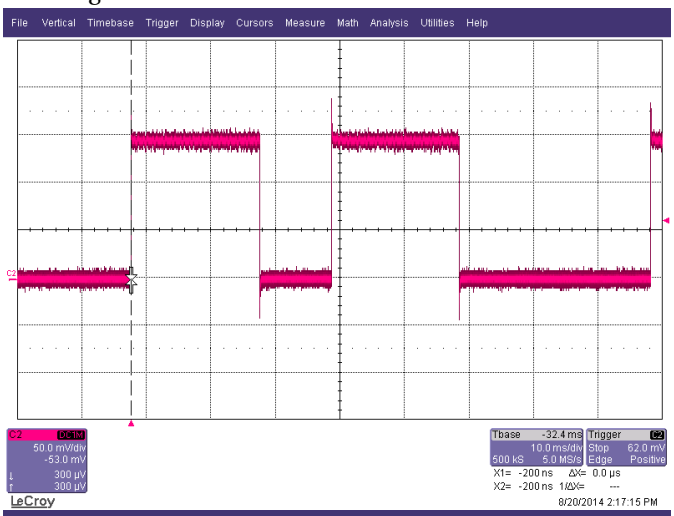


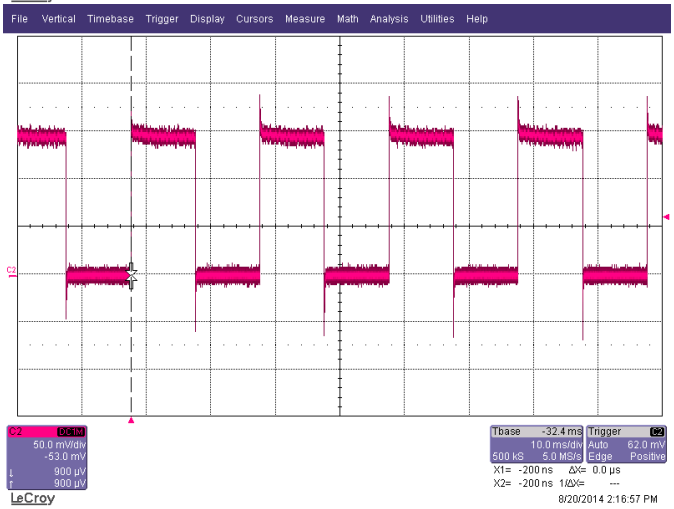
Sans charge

7.2 Attente par *timer*, sans Xenomai

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#include "gpio_lib.h"
#define PG9     SUNXI_GPG(9)

#define TIMESLEEP 500

int status=0; // declenche' a chaque appel du timer => exterieur

void test(int signum) {
    sunxi_gpio_output(PG9,status);
    status^=0xff;
}

int main()
{int i,status=0;
 struct sigaction sa;
 struct itimerval timer;

 if(sunxi_gpio_init())                  {printf("Failed init  \n");return -1;}
 if(sunxi_gpio_set_cfgpin(PG9,OUTPUT)) {printf("Failed config\n");return -1;}

 memset(&sa, 0,sizeof(sa));
 sa.sa_handler = &test;
 sigaction(SIGVTALRM,&sa, NULL);
 /* Configure the timer to expire after TIMESLEEP msec ... */
 timer.it_value.tv_sec = 0;
 timer.it_value.tv_usec = TIMESLEEP;
 /* ... and every TIMESLEEP usec after that. */
 timer.it_interval.tv_sec = 0;
 timer.it_interval.tv_usec = TIMESLEEP;
 setitimer(ITIMER_VIRTUAL, &timer, NULL);
 while(1);
 sunxi_gpio_cleanup();
 return 0;
}
```

En charge



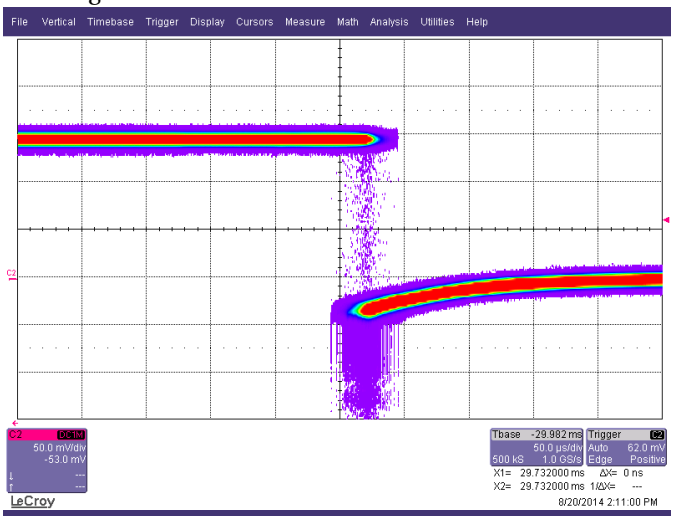


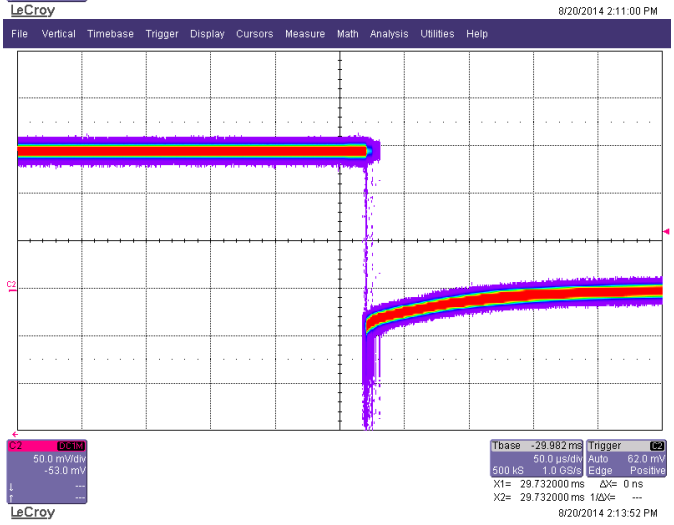
Sans charge

7.3 Attente par `sleep`, avec Xenomai

```
1  #include <signal.h>
2
3  #include <native/task.h>
4  #include <native/timer.h>
5  #include <rtdm/rtdm.h>
6  #include <sys/types.h>
7  #include <sys/stat.h>
8  #include <fcntl.h>
9  #include <unistd.h>
10 #include <sys/mman.h>
11 #include "gpio_lib.h"
12 #define PG9     SUNXI_GPG(9)
13
14 #define TIMESLEEP 500000
15
16 RT_TASK blink_task;
17
18 void blink(void *arg){
19  int status=0;
20  struct timespec tim = {0,TIMESLEEP};
21
22  while(1)
23    {sunxi_gpio_output(PG9,status);
24     status^=0xff;
25     if (nanosleep(&tim,NULL) != 0)
26        {printf("erreur usleep\n");return;}
27    }
28 }
29
30 void catch_signal(int sig){}
31
32 int main(int argc, char **argv) {
33  if(sunxi_gpio_init())                    {printf("Failed →
       ↪init  \n");return -1;}
34  if(sunxi_gpio_set_cfgpin(PG9,OUTPUT)) {printf("Failed →
       ↪config\n");return -1;}
35  signal(SIGTERM, catch_signal);
36  signal(SIGINT, catch_signal);
37  /* Avoid memory swapping for this program */
38  mlockall(MCL_CURRENT|MCL_FUTURE);
39  rt_task_create(&blink_task, "blinkLed", 0, 99, 0);
40  rt_task_start(&blink_task, &blink, NULL);
41
42  pause();
43
44  rt_task_delete(&blink_task);
45  return 0;
46 }
```

En charge



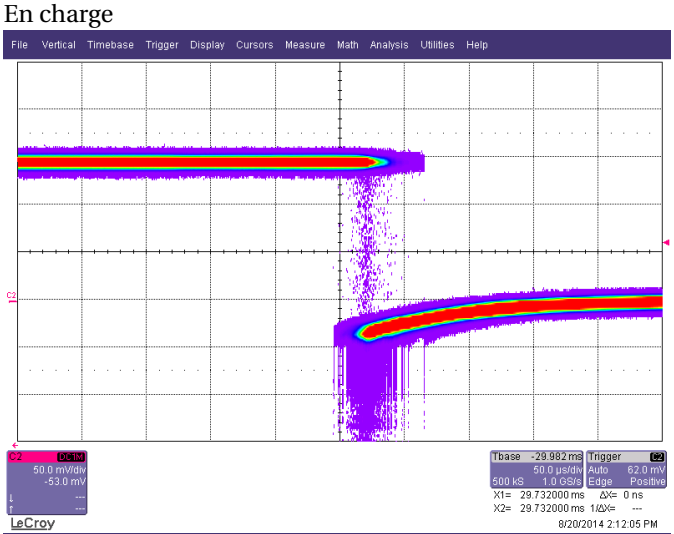


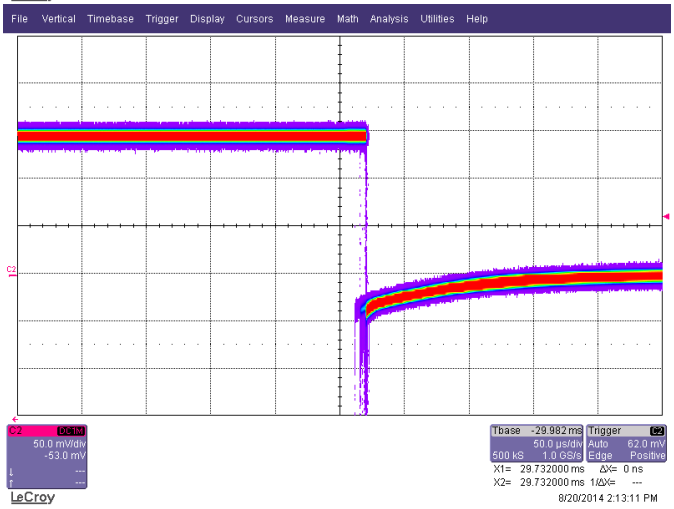
Sans charge

7.4 Attente par *timer*, avec Xenomai

```c
#include <signal.h>
#include <native/task.h>
#include <native/timer.h>
#include <rtdm/rtdm.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include "gpio_lib.h"
#define PG9     SUNXI_GPG(9)

#define TIMESLEEP 500000

RT_TASK blink_task;

void blink(void *arg){
 int status=0;
 struct timespec tim = {0,TIMESLEEP};

 rt_task_set_periodic(NULL, TM_NOW, TIMESLEEP*1000);
 while(1)
   {rt_task_wait_period(NULL);
    sunxi_gpio_output(PG9,status);
    status^=0xff;
   }
}

void catch_signal(int sig){}

int main(int argc, char **argv) {
 if(sunxi_gpio_init())                    {printf("Failed →
     ↪init  \n");return -1;}
 if(sunxi_gpio_set_cfgpin(PG9,OUTPUT)) {printf("Failed →
     ↪config\n");return -1;}
 signal(SIGTERM, catch_signal);
 signal(SIGINT, catch_signal);
 mlockall(MCL_CURRENT|MCL_FUTURE); // Avoid memory →
     ↪swapping for this program
 rt_task_create(&blink_task, "blinkLed", 0, 99, 0);
 rt_task_start(&blink_task, &blink, NULL);
 pause();
 rt_task_delete(&blink_task);
 return 0;
}
```

En charge





Sans charge

A Portage de Xenomai, sur OlinuxIno A13 micro, dans buildroot

La mise en œuvre de Xenomai sur un processeur particulier passe par la présence, pour cette architecture, du support de **Ipipe**. En effet, pour autoriser l'exécution en parallèle de deux noyaux (l'un temps-réel, l'autre temps-partagés), avec une garantie de latence minimale, il est nécessaire d'ajouter une couche d'abstraction entre le matériel et le logiciel. Celle-ci va mettre à disposition, dans l'ordre des priorités, les interruptions matérielles, et éviter qu'un des domaines ne se trouve bloqué sur l'attente d'une interruption masquée par l'autre domaine.

Le problème qui se pose avec les processeurs allwinner est qu'il n'existe pas, à l'heure actuelle, de support officiel.

Toutefois, Paul Crubille [5] a réalisé ce travail sur noyau sunxi 3.4.24 et le propose au travers de deux patchs :

1. **pre.patch** qui supprime les modifications apportées entre la version 3.4.6 (version sur lequel Ipipe s'applique) et la version 3.4.24 du noyau linux (en tenant compte également des modifications entre un noyau officiel et celui proposé par sunxi ;
2. **post.patch** qui re-ajoute les modifications supprimées par **pre.patch** ainsi que le support Ipipe pour a13.

Il est ainsi possible, de manière manuelle, d'appliquer **pre.patch**, de lancer **prepare-kernel** fournit dans l'archive de Xenomai en lui spécifiant le patch Ipipe adapté et finalement d'appliquer **post.patch** afin d'obtenir un noyau avec le support temps-réel pour les processeurs A13.

5. http://www.xenomai.org/pipermail/xenomai/2013-February/027801.html

A.1 Création d'un patch Ipipe avec le support de l'A13

La solution en trois patchs (pre, prepare et post), dans le cadre de l'utilisation de buildroot, n'est pas adaptée. Buildroot ne fournit pas de solutions pour une telle manipulation et ne prends qu'un seul patch qui est censé ajouter le support d'Ipipe. Afin de disposer de Xenomai sur l'A13, dans buildroot, pour la génération d'une image de manière automatique (sans interventions manuelles) la solution consiste donc à re-générer le patch Ipipe pour le noyau 3.4.24 avec le support du processeur cible.

Ceci va se faire en 4 étapes :

- récupérer le noyau sunxi-v3.4.24-r2 [6] ;
- le décompresser et en faire une copie de référence (non modifiée) ;
- se placer dans le répertoire du noyau à modifier et appliquer les trois patchs, dans l'ordre, à l'aide de la commande

```
patch -p1 < xxx.patch
```

- réaliser le diff entre la version non modifiée et la version sur lesquels les patchs ont étés appliqués à l'aide de la commande :

```
diff -ruN linux.orig linux > a13-ipipe-core-3.4.6-arm-4.patch
```

(avec 'r' pour récursif, 'u' pour unifié (nécessaire pour la commande patch) et 'N' pour que **diff** affiche le contenu complet du fichier s'il est absent d'un des deux répertoires au lieu de simplement signaler qu'il n'est présent que dans l'un d'eux).

Cette série d'étapes a permis de disposer d'un fichier a13-ipipe-core-3.4.6-arm-4.patch équivalent à celui fourni par le projet Xenomai mais avec le support pour l'A13 et applicable directement sur le noyau 3.4.24 sunxi.

Ce fichier, par commodité, peut être déplacé dans `board/a13_olinuxino/` d'une copie locale de buildroot.

A.2 Configuration de Buildroot et réalisation d'un defconfig dédié

Ayant un patch adapté au noyau 3.4.24 de sunxi avec le support pour l'olinuxino a13 micro, l'étape suivante consiste à créer un fichier de configuration pour l'environnement buildroot permettant la génération automatique d'une image avec le support temps-réel.

Le dépot fournit un support pour l'a13 sans Xenomai. La première étape consiste à configurer l'environnement par : `make a13_olinuxino_micro_defconfig`.

Ensuite, cette configuration par défaut doit être adaptée aux contraintes liées à la mise en place de Xenomai.

Dans l'interface de configuration de buildroot les modifications sont les suivantes :

Parties noyau, sous menu `Kernel --> :`, les opérations sont

1. par défaut le noyau à télécharger est un dépot git :
   - Kernel Version qui est fixé à `Custom Git repository` doit être modifié en `custom tarball`
   - URL of custom kernel tarball doit être renseigné avec la valeur :

```
https://github.com/linux-sunxi/linux-sunxi/archive/sunxi-v3.4.24-r2.tar.gz
```

2. le support pour l'extension adeos/Ipipe doit être activée en spécifiant le chemin relatif pour le patch généré précédemment :

```
Kernel -->
  Linux Kernel Extensions -->
    [*] Adeos/Xenomai Real-Time patch
    (board/a13_olinuxino/a13-ipipe-core-3.4.6-arm-4.patch) Path for Adeos patch file)
```

Parties applications :

Xenomai et ses options de bases sont activées par défaut (sélectionnées par l'activation de Adeos dans le menu kernel), seul testsuite doit être activée manuellement :

```
Dans Target packages -->
  Real-Time -->
    Xenomai Userspace
    [*] Install testsuite
    [*] Native skin library
    [*] Posix skin library
```

Les modifications dans la configuration de buildroot étant finies, la dernière étape consiste à produire le fichier de configuration par défaut (defconfig) correspondant :

Après être sortie de l'interface de configuration, la commande

`make savedefconfig` va produire un fichier defconfig. Ce fichier doit être copié en `configs/a13_olinuxino_micro_xenomai_de`

6. https://github.com/linux-sunxi/linux-sunxi/archive/sunxi-v3.4.24-r2.tar.gz

Références

[1] P.Ficheux, *Les distributions "embarquées" pour Rasberry PI*, Opensilicium 7 (2013)

[2] P. Kadionik, P.Ficheux, *Temps réel sous LINUX (reloaded)*, GNU/Linux Magazine France HS 24 (2006), disponible à `http://pficheux.free.fr/articles/lmf/hs24/realtime/linux_realtime_reloaded_final_www.pdf` et `http://www.unixgarden.com/index.php/gnu-linux-magazine-hs/temps-reel-sous-linux-reloaded`

[3] C. Blaess, *Solutions temps réel sous Linux*, Eyrolles (2012)