N°47 AVRIL MAI 2010

France Métro : 6,50 € / DOM : 7 €
TOM Surface : 950 XPF / POL. A :1400 XPF
CH : 13,80 CHF / BEL,PORT.CONT : 7,50 €
CAN : 13 $CAD / TUNISIE : 8,80 TND / MAR : 75 MAD

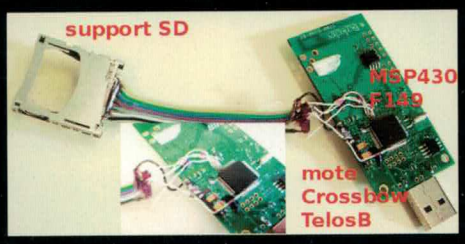
GNU LINUX MAGAZINE / FRANCE

HORS-SÉRIE

Administration et développement sur systèmes UNIX

SD FAT / MSP430

Développez un système d'exploitation pour microcontrôleur MSP430 : stockage de masse en FAT sur carte SD



FPGA / SOFTCORE

Mise en œuvre de Linux sur le processeur softcore libre LEON

DISTRIBUTION

Découvrez Buildroot, un environnement de construction de systèmes Linux pour l'embarqué

BUILDROOT

Cas pratique de mise en œuvre de Buildroot sur carte ARM9 DEV2410

SPÉCIAL EMBARQUÉ

APPLICATIONS / SYSTÈMES / MATÉRIELS

VOYAGE AU CENTRE DE L'EMBARQUÉ...

ANDROID, SYMBIAN
OPENWRT, BUILDROOT
MSP430, SOFTCORE LEON, ...

PYTHON / NOKIA

Développer des applications pour mobile n'a jamais été aussi simple !

ANDROID / JAVA

Découvrez comment développer sur le système d'exploitation pour smartphone de Google

GPS / GOOGLE MAP

Complétez votre application Android avec le positionnement GPS et l'accès aux services en ligne

OPENWRT / ACME

Installation d'OpenWrt Kamikaze trunk sur carte ACME Fox Board Classic 416

L 15066 - 47 H - F: 6,50 € - RD

SOMMAIRE

CRÉEZ UNE APPLICATION

CONSTRUISEZ UN SYSTÈME

DÉVELOPPEZ POUR LE MATÉRIEL

ABONNEMENT

ÉDITO

L'embarqué est un vaste domaine. D'un point de vue strict, un système embarqué peut être défini comme un ensemble électronique et informatique autonome, généralement dédié à une tâche précise, dont les ressources sont souvent limitées (encombrement, puissance, consommation, etc.). Cette courte définition est-elle encore valable aujourd'hui ? Avec la montée en puissance des architectures à base de dernières générations d'ARM et les avancées non négligeables en termes de ressources mémoire et d'économie d'énergie, les cartes/kits d'évaluation ressemblent de plus en plus à des PC complets.

Il devient ainsi très difficile de rester dans une logique où tout l'embarqué peut être envisagé d'un seul tenant. Nous avons donc choisi, dans ce numéro, de dégager trois grands axes de développement du domaine.

D'une part, nous avons le développement d'application reposant intégralement sur les fonctionnalités mises à disposition par le système et le matériel sous-jacent. L'exemple le plus démonstratif est, sans le moindre doute, la création d'applications pour smartphones Androïd, sujet de choix traité par Marc de Verdelhan des Molles.

Si nous descendons d'une couche, nous touchons au système à proprement parler. Pierre Ficheux nous offre ainsi une étude complète ainsi qu'un voyage captivant dans Buildroot, le constructeur de systèmes embarqués basés sur Linux. Au terme de la lecture de ce double article, vous serez à même de porter Linux et les outils système vers des plates-formes encore inexplorées.

Enfin, toujours plus près du matériel, nous arrivons à un niveau où l'abstraction devient presque formelle. C'est le monde des microcontrôleurs qui, eux aussi, gagnent sans cesse en puissance et en ressources. Jean-Michel Friedt et Gwenhaël Goavec-Merou nous proposent ainsi la mise en œuvre et l'exploitation d'un OS pour le MSP430 de Ti.

Paradoxalement, l'embarqué prend de plus en plus de place dans l'univers du logiciel libre et ce hors-série, je l'espère, vous convaincra de l'étendue et la diversité des possibilités qui s'offrent à vous. Vous n'avez finalement plus qu'à choisir le niveau d'abstraction qui vous convient avec le matériel...

Denis Bodor

Sorcier niveau 18

GNU/Linux Magazine est membre de l'APRIL

Gnu /Linux Magazine France Hors-série
est édité par Les Editions Diamond

LES ÉDITIONS DIAMOND

B.P. 20142 - 67603 Sélestat Cedex
Tél. : 03 67 10 00 20
Fax : 03 67 10 00 21
E-mail : lecteurs@gnulinuxmag.com
Service commercial : abo@gnulinuxmag.com
Sites : www.gnulinuxmag.com
www.ed-diamond.com
Directeur de publication : Arnaud Metzler
Rédacteur en chef : Denis Bodor
Secrétaire de rédaction : Véronique Wilhelm
Relecture : Véronique Wilhelm
Conception graphique : Kathrin Troeger
Responsable publicité : Tél. : 03 67 10 00 26
Service abonnement : Tél. : 03 67 10 00 20
Impression : VPM Druck Allemagne
Distribution France :
(uniquement pour les dépositaires de presse)
MLP Réassort :
Plate-forme de Saint-Barthélemy-d'Anjou.
Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier.
Tél. : 04 74 82 63 04
Service des ventes :
Distri-médias :
Tél. : 05 34 52 34 01
IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : À parution, **N° ISSN :** 1291-78 34
Commission paritaire : K78 976
Périodicité : Bimestrielle
Prix de vente : 6,50 €

Ma première application

De tous les systèmes d'exploitation libres pour mobile, Android est aujourd'hui celui dont la part de marché croît le plus vite. De fait, il devient intéressant de proposer des exemples d'applications qu'il est possible de développer sur cette plate-forme. C'est ce que nous nous proposons de faire dans cet article.

Auteur

- Marc de Verdelhan des Molles

1 PRÉAMBULE

L'application que nous nous proposons de développer est relativement simple. Plus complexe qu'un « Hello World ! », elle vise cependant à faire découvrir au lecteur certains concepts inhérents au développement sous Android. Le service qu'elle rend consiste à présenter à l'utilisateur un écran comportant trois boutons, chacun correspondant à un rayon (en km) de recherche. Lorsque l'on clique sur un de ces boutons, l'application géolocalise l'appareil et affiche une *Google Map* comportant les emplacements des déchetteries **[DECHETTERIES]** dans le rayon souhaité.

La description du développement de ce programme est divisée en deux parties, chacune faisant l'objet d'un article. Dans ce premier volet, nous mettrons en place l'environnement de développement ; nous traiterons de la création du premier écran ; nous introduirons les concepts de *layout*, d'*activité*, d'*intent* ; enfin, nous verrons comment passer d'un écran à l'autre au sein de l'application. La seconde partie de notre étude portera sur l'interrogation du service web à l'aide de la méthode SAX et sur le positionnement des objets sur la Google Map.

1.1 Initialisation du projet

On part du principe que le lecteur possède déjà l'environnement de développement suivant :

- Eclipse 3.5 (Galileo) ;
- le SDK Android 1.6 r1 **[ANDROID_1.6]** ;
- le plugin Eclipse *Android Development Tools (ADT)* (version 0.9.3).

On suppose qu'Eclipse est configuré de telle façon qu'il connaît le chemin des outils du SDK.

Pour plus d'informations sur ce dont on a besoin, je vous invite à vous rendre sur le site de Google **[INSTALL_IDE]**, où tout est expliqué.

Pour pouvoir tester l'application dans l'émulateur, il est d'abord nécessaire de créer un AVD (*Android Virtual Device*). Pour ce faire, on commence par lister les cibles possibles à l'aide de la commande `android list targets`. On choisit la cible compatible Android 1.6 et possédant la bibliothèque des cartes Google dont nous nous servirons tout à l'heure.

```
android create avd --target 4 --name my_avd
```

Maintenant que l'environnement est fin prêt, nous pouvons créer le projet. Pour ce faire, exécutez Eclipse et cliquez sur ***File >> New >> Project... >> Android >> Android Project***. Cliquez sur ***Next***. On appelle notre projet « MyFirstApp » (champ ***Project name***). Dans ***Build Target***, sélectionnez « Google APIs » dans sa version 1.6. On notera que le champ ***Min SDK Version*** prend alors la valeur 4. Cette valeur indique le niveau de l'API minimum requis par notre application. Un smartphone Android ne supportant qu'un niveau d'API inférieur à ce ***Min SDK Version*** n'acceptera pas l'installation de notre programme. Ici, on cible donc les équipements Android 1.6 **[BASE]**. Remplissez maintenant le champ ***Application name*** avec « My First App » et le champ ***Package name*** avec « com.android.myfirstapp ». Laissez cochée l'option ***Create Activity*** et remplissez le champ adjacent avec « MyFirstActivity ». Cliquez sur ***Finish***. Le projet est créé.

Dans le ***Package Explorer***, on peut voir que ce projet comporte - outre les .jar de l'API Google - trois dossiers. Le dossier `src/` contient les sources Java (dont la première classe : `MyFirstActivity.java`) de notre application. Le dossier `res/` contient les ressources (icônes, layouts, ...) liées au projet. Enfin, le répertoire `gen/` contient des fichiers sources Java mis à jour automatiquement par l'environnement de développement Android.

Android - 1ère partie

2 PREMIER ÉCRAN : LE MENU

Dans un projet Android, un écran d'action correspond à une activité. Notre projet prévoit deux écrans (le menu et la carte) et donc deux activités. Une activité est une classe héritant de la classe `Activity`. Comme vous l'aurez sans doute deviné, la classe `MyFirstActivity` est la première activité de notre programme. Nous allons y dessiner le menu.

`MyFirstActivity` comporte une méthode publique `onCreate()`. Cette méthode est appelée implicitement lors de la création de l'activité, c'est-à-dire qu'elle s'exécute lorsque l'écran correspondant apparaît à l'utilisateur. Au stade où nous en sommes, nous n'y trouvons que `setContentView(R.layout.main);`. Ce dernier appel de fonction indique au système qu'il doit utiliser le *layout* `main` pour cette première activité.

2.1 Layout du menu

Le layout d'une activité est un fichier permettant de spécifier l'organisation des composants graphiques de l'écran correspondant. Pour savoir quel layout utiliser dans une activité, on passe un identifiant (`int`) à `setContentView`. La valeur de cet entier se trouve dans la classe `R` (Voir le fichier `gen/R.java`) générée automatiquement. L'ajout de cet identifiant à la classe `R` a été réalisé lorsque le layout `main` a été créé. Ce dernier l'a été avec la classe `MyFirstActivity` et donc en même temps que la création du projet. Il est décrit sous la forme d'un fichier XML à l'emplacement `res/layout/main.xml`.

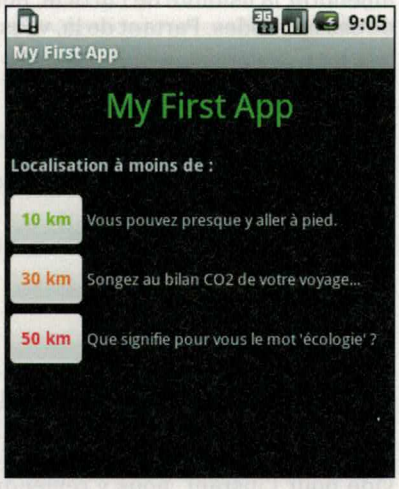


Fig. 1 : Représentation graphique de la première activité (menu) de l'application

Voici à quoi l'on veut faire ressembler le premier écran de notre programme.

Pour cela, ouvrez le fichier `res/layout/main.xml`. Normalement, vous voyez apparaître un écran qui vous présente ce que vous verriez si vous exécutiez le programme dès maintenant. En l'occurrence, il s'agirait d'un écran noir avec écrit « Hello World, MyFirstActivity! ». Cette interface vous permet de concevoir graphiquement votre layout. La vue ***Outline*** vous présente l'arborescence des composants/layouts utilisés pour cet écran.

Juste sous l'écran noir, on trouve deux onglets. L'un permet justement la conception graphique de l'écran (***layout***), l'autre donne l'équivalent en XML (***main.xml***). Dans le cadre de cet article, on s'intéressera essentiellement au second onglet, ceci afin de simplifier l'élaboration de nos layouts. Cliquez donc sur ***main.xml*** afin de trouver le code XML correspondant. Comme on l'avait vu dans la fenêtre ***Outline***, on trouve un layout de type *LinearLayout* **[LAYOUT]**, à orientation verticale (`android:orientation="vertical"`) et contenant un objet TextView. Le TextView en question affiche le texte contenu dans sa propriété `android:text`. Ici, il s'agit de `@string/hello`, qui est une chaîne constante.

Chaque application Android possède des chaînes de caractères constantes. Ces constantes sont listées dans le fichier XML `res/values/strings.xml`. Ouvrez ce dernier fichier, vous pouvez voir deux constantes. L'une s'intitule `hello`, l'autre correspond au nom de votre application. Sélectionnez la constante `hello`, vous pouvez voir sa valeur. Comme nous n'en aurons pas besoin (à moins que vous ne teniez particulièrement à votre Hello World), vous pouvez cliquer sur le bouton ***Remove...*** pour la supprimer.

Une fois cela fait, enregistrez le `strings.xml` et retournez au fichier de layout `main.xml`. Vous remarquerez qu'Eclipse y trouve une erreur. Normal, la constante qu'il utilisait vient d'être supprimée.

Pour simplifier, on commence par vider `main.xml` de tout son contenu et on le remplace par le code que l'on décrit maintenant.

```
<?xml version="1.0" encoding="utf-8"?>

<RelativeLayout android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    xmlns:android="http://schemas.android.com/apk/res/android">
```

On utilise un RelativeLayout. Ce n'est pas le plus simple, mais c'est a priori ce qu'il y a de plus flexible. Il permet en effet de placer chaque composant relativement aux autres objets de l'écran. On précise que ce layout doit remplir le composant parent (`fill_parent`). En l'occurrence, il s'agit du layout Android de base.

```
<TextView android:id="@+id/titre"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="10sp"
    android:text="@string/app_name"
    android:textColor="#0f0"
    android:textSize="28sp">
</TextView>
```

Dans le RelativeLayout, on commence par afficher le titre. On utilise pour cela un composant TextView. On lui donne un *id* (`titre`). On précise que sa taille doit s'adapter à son contenu (`wrap_content`). Les autres paramètres sont suffisamment transparents pour qui possède des notions de base de CSS **[TEXTVIEW]**. La valeur de ce composant texte est bien entendu précisée par l'attribut `text`. On notera qu'ici, il s'agit de la chaîne constante correspondant au nom de l'application.

```
<TextView android:id="@+id/texteRayon"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text=" Localisation à moins de :"
    android:layout_below="@+id/titre"
    android:layout_marginBottom="12sp"
    android:textStyle="bold"
    android:layout_marginTop="20sp">
</TextView>
```

On poursuit avec le texte situé entre le titre et les boutons. On utilise un autre TextView, que l'on appelle `texteRayon`. On insistera juste sur l'attribut `layout_below`, qui permet de spécifier en dessous de quel composant doit être placé ce texte. Ici, il s'agit de l'objet titre (`layout_below="@+id/titre"`). Passons maintenant aux boutons de rayon.

```
<Button android:id="@+id/btn10km"
    android:layout_height="wrap_content"
    android:layout_width="wrap_content"
    android:textStyle="bold"
    android:text="10 km"
    android:textColor="#7f0"
    android:layout_below="@+id/texteRayon"
    android:width="30pt">
</Button>
```

On crée le bouton des « 10 km ». On le place sous le texte précédent (`layout_below="@+id/texteRayon"`).

```
<TextView android:id="@+id/texte10km"
    android:layout_height="wrap_content"
    android:layout_width="wrap_content"
    android:layout_alignBaseline="@+id/btn10km"
    android:text="Vous pouvez presque y aller à pied."
    android:layout_toRightOf="@+id/btn10km"
    android:textSize="13sp">
</TextView>
```

On place un petit texte explicatif à côté du bouton. Ici, ce texte est aligné sur la même ligne (`layout_alignBaseline`) et à droite (`layout_toRightOf`) du bouton `btn10km`.

Vous pouvez maintenant reproduire les deux autres boutons en les positionnant en dessous les uns des autres. Une fois que cela est fait, il ne nous reste plus qu'à fermer le RelativeLayout : `</RelativeLayout>`.

Enregistrez le fichier `main.xml`, passez en mode de conception graphique. Normalement, vous devriez voir votre layout tel qu'il est représenté sur la figure 1. En outre, on pourra noter qu'en cas de modification d'un composant pour lequel a été précisé un identifiant, la classe `R` est automatiquement mise à jour lors de l'enregistrement du fichier de layout.

2.2 Activité du menu

La création de notre première activité continue. Maintenant que nous avons produit le layout, il nous faut l'utiliser. Cela se fait dans la classe de l'activité, ici `MyFirstActivity.java`.

On commence en créant les attributs dont nous avons besoin.

```
private LocationManager lm; /** Location manager */
/** Boutons */
private Button btn10km;
private Button btn30km;
private Button btn50km;
```

On ajoute les trois boutons de rayon et le LocationManager permettant la gestion du service de localisation.

```
private class MyLocationListener
          implements LocationListener {

}
```

On ajoute la classe privée `MyLocationListener` servant à recevoir les notifications de la part du LocationManager lors des changements de position. Une fois les `import` correspondants ajoutés, il reste une erreur. Cliquez dessus et sélectionnez ***Add unimplemented methods***. Pour simplifier et améliorer la lisibilité de l'article, nous ne nous servirons pas de ces méthodes. Partant de là, vous pouvez les laisser vides.

En revanche, nous avons besoin d'un listener de clic sur les boutons :

```
/** Listener sur les boutons */
private OnClickListener btnListener = new OnClickListener()
{
    /**
     * @param arg0 "vue" sur laquelle on a cliqué
     */
    public void onClick(View arg0) {
        // TODO Auto-generated method stub
    }
};
```

Ce listener possède bien évidemment une méthode `onClick()` (qui s'exécutera lorsque l'utilisateur cliquera sur un bouton auquel on aura associé le listener). Laissons-la vide pour l'instant, nous y reviendrons tout à l'heure.

Enfin, il nous faut une méthode qui, à partir du bouton sur lequel on a cliqué, retourne le rayon (nombre de kilomètres) correspondant. Cette méthode prend la forme suivante :

```
/**
 * @param btn bouton à partir duquel on recupère le rayon
 * @return le rayon
 */
private int getRayon(View btn) {
    if(btn10km.equals(btn)) {
        return 10;
    } else if(btn30km.equals(btn)) {
        return 30;
    } else if(btn50km.equals(btn)) {
        return 50;
    } else {
        Log.d("Debug MyFirstApp", "Bouton inattendu");
        return 10; // Rayon par défaut
    }
}
```

Les préliminaires sont terminés : on passe à l'exécution de la méthode **onCreate()**. Sous l'appel à **setContentView()**, on initialise les attributs de la classe :

```
// Création des boutons de rayon
btn10km = (Button) findViewById(R.id.btn10km);
btn10km.setOnClickListener(btnListener);
btn30km = (Button) findViewById(R.id.btn30km);
btn30km.setOnClickListener(btnListener);
btn50km = (Button) findViewById(R.id.btn50km);
btn50km.setOnClickListener(btnListener);
```

La méthode **findViewById()** retourne la vue (**View**) correspondant à l'identifiant passé en paramètre. On *cast* cette vue pour initialiser les attributs « boutons » de la classe. Puis on associe à chacun d'entre eux le OnClickListener créé plus haut. Les boutons terminés, on peut passer au LocationManager :

```
// Récupération du LocationManager
this.lm = (LocationManager)
          getSystemService(Context.LOCATION_SERVICE);
this.lm.requestLocationUpdates(
          "gps", 0, 0, new MyLocationListener());
```

Android considère les périphériques matériels (GPS, accéléromètre, réseau, ...) comme des services. Ici, on récupère un LocationManager qui n'est autre qu'un objet permettant d'utiliser le service de géolocalisation (**LOCATION_SERVICE**). On demande ensuite à être notifié (**requestLocationUpdates()**) de tout changement dans la position géographique du terminal Android.

Ça y est ! Le squelette de notre première activité est en place. Afin qu'elle soit utile à quelque chose, il faut maintenant implémenter le contenu de la méthode **onClick()** du listener appliqué aux boutons. Pour ce faire, remplacez la ligne **// TODO Auto-generated method stub** qu'elle contient par le code suivant :

```
/**
 *  Récupération de la position depuis
 *  le "location provider" GPS
 */
Location loc;
loc = MyFirstActivity.this.lm.getLastKnownLocation("gps");
if(loc == null) {
    /**
     *  Pas de GPS ?
     *  Récupération de la position depuis
     *  le "location provider" réseau
     */
    loc = MyFirstActivity.this.lm.getLastKnownLocation("network");
}
```

Cet extrait a pour objectif de récupérer la dernière position connue du terminal Android. On utilise pour cela la méthode **getLastKnownLocation()** du LocationManager défini précédemment. Cette position est en général obtenue via le service de GPS, mais en cas de défaillance de ce dernier, elle peut l'être depuis le service réseau. Cette phase de géolocalisation aurait également pu être exécutée au chargement de l'activité (dans la méthode **onCreate()**). L'inconvénient de cette solution est qu'elle ne prend pas en compte les éventuels déplacements qui pourraient être effectués entre le lancement du programme et le clic de l'utilisateur sur un bouton (NDLR : ceci peut paraître logique, mais il est tout de même important de bien garder à l'esprit qu'on développe pour un périphérique mobile). C'est pourquoi on préfère l'exécuter dans la méthode **onClick()**.

A la suite du code précédent, on ajoute les lignes qui suivent :

```
if(loc != null) {
    // Lyonnais
    //loc.setLatitude(45.807502);
    //loc.setLongitude(4.806687);
    /**
     * Lancement de l'activité MyMap et passage des
     * paramètres
     */
    Intent intent;
    intent = new Intent(MyFirstActivity.this, MyMap.class);
    Bundle bundle = new Bundle();
    bundle.putDouble("LATITUDE", loc.getLatitude());
    bundle.putDouble("LONGITUDE", loc.getLongitude());
    bundle.putInt("RAYON", getRayon(arg0));
    intent.putExtras(bundle);
    startActivity(intent);
} else {
    // Impossible de récupérer la position
    Toast.makeText(MyFirstActivity.this,
            "Impossible de récupérer la position",
            Toast.LENGTH_SHORT).show();
}
```

Ici, on teste une nouvelle fois si on a finalement pu récupérer la position. Si ce n'est pas le cas, on affiche un court message à l'utilisateur, lui indiquant qu'il n'a pas été possible de le faire. En revanche, si la géolocalisation a fonctionné, on poursuit l'exécution et on passe à l'activité suivante.

Concrètement, comment se déroule cette dernière phase ? On crée un *intent* **[INTENT]** qui correspond à une action à exécuter (ici, le lancement d'une nouvelle activité). On l'initialise notamment avec la classe de l'activité à lancer. Comme cette dernière n'est pas encore créée, on décide de l'appeler « MyMap » (**MyMap.class**). A cet intent, on fournit des paramètres qui seront transmis à MyMap. Ces paramètres sont stockés dans un objet **Bundle**, qui n'est en réalité rien d'autre qu'un tableau associatif évolué, et que l'on associe à l'intent à l'aide de la méthode **putExtras()**. On lance ensuite la nouvelle activité en appelant **startActivity()**.

Enfin ! La première activité est terminée. Notre projet comporte maintenant une erreur. Elle se trouve sur le **MyMap.class**, qui est une référence à la future seconde activité. Passons à la création de cette dernière pour qu'Eclipse se rendorme.

3 DEUXIÈME ÉCRAN : LA CARTE

Comme la première, la seconde activité nécessite un layout et un fichier Java.

Pour ce qui est du layout, il suffit de faire un ***clic droit sur le dossier res/layout >> New >> File***. On rentre alors « map.xml » dans ***File name*** et on clique sur ***Finish***. On copie enfin le code suivant dans le fichier nouvellement créé (***res/layout/map.xml***) :

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    xmlns:android="http://schemas.android.com/apk/res/android">
    <com.google.android.maps.MapView
        android:id="@+id/gmapview"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:enabled="true"
        android:clickable="true"
        android:apiKey="PUT_HERE_THE_API_KEY"/>
</RelativeLayout>
```

Ce second layout est uniquement composé d'une **MapView** permettant la visualisation de la Google Map. Le seul point obscur de ce code concerne l'*API key*. La classe **MapView** donne accès à des données fournies par Google Maps. Pour pouvoir nous en servir, il faut accepter les *Terms of Service* de Google, sans quoi votre MapView ne vous présentera qu'un écran vide. L'API key sert à vous identifier en tant que développeur ayant bien accepté les conditions d'utilisation du service. Pour la récupérer, je vous invite à vous rendre sur cette page **[API_KEY]** et à suivre la procédure qui y est décrite. Une fois obtenue, vous pouvez l'utiliser pour remplacer **PUT_HERE_THE_API_KEY**.

Passons maintenant au fichier Java correspondant. Au même emplacement que **MyFirstActivity.java**, créez une nouvelle classe du nom de **MyMap.java** et remplacez le peu de code qu'elle comporte par celui-ci :

```
package com.android.myfirstapp;

public class MyMap extends MapActivity {

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

    }

    @Override
    protected boolean isRouteDisplayed() {
        // TODO Auto-generated method stub
        return false;
    }
}
```

Ajoutez les **import** et sauvegardez votre fichier. Comme la première activité, celle-ci comporte une méthode **onCreate()**. En revanche, cette classe n'hérite pas d'**Activity** mais de **MapActivity**. Cela l'oblige à redéfinir la méthode **isRouteDisplayed()** (voir les spécifications de la classe MapActivity **[SPEC_MAPACT]** pour plus d'informations sur cette méthode). L'enregistrement de ce fichier corrige maintenant l'erreur qu'Eclipse signalait au niveau du **MyMap.class** dans l'activité précédente. A cette nouvelle classe, on ajoute des attributs.

```
private MapView mapView; /** Vue de la carte */
private MapController mc; /** Controleur de la carte */
private int zoomInitial; /** Valeur initiale du zoom */
private GeoPoint centreInitial; /** Centre de la carte */
```

Ici, je pense que les commentaires sont suffisamment explicites pour que l'on puisse se passer d'explications.

On passe à la méthode **onCreate()**, à laquelle on ajoute ces derniers bouts de code :

```
setContentView(R.layout.map);
mapView = (MapView) findViewById(R.id.gmapview);
mc = mapView.getController();
```

On précise qu'on veut utiliser le layout **map** pour cette activité. On associe la vue **gmapview** à l'objet **mapView** (de type **MapView**, comme ça c'est clair !). On récupère également le contrôleur de l'objet **mapView**.

```
// Récupération des paramètres
Bundle bundle = this.getIntent().getExtras();
double latitude = bundle.getDouble("LATITUDE");
double longitude = bundle.getDouble("LONGITUDE");
int rayon = bundle.getInt("RAYON");
```

Vous vous souvenez que dans la première activité, nous avions passé à l'intent un *bundle* contenant les paramètres à fournir à l'activité suivante. Ici, on récupère ces paramètres dans notre nouvelle activité.

```
zoomInitial = 14;
centreInitial = new GeoPoint((int) (latitude * 1E6),
                             (int) (longitude * 1E6));
```

Le zoom initial correspond à la valeur de zoom que l'on veut avoir pour la visualisation de la carte au chargement de l'activité. Plutôt que de le définir en dur, on aurait également pu le calculer en fonction du rayon choisi par l'utilisateur (on peut aisément comprendre l'intérêt d'avoir un champ de vision plus large si l'on augmente le rayon de recherche). Le centre initial est le point géographique que l'on veut avoir au centre de l'écran lors de l'affichage de la carte. Pour sa part, ce dernier point est bien évidemment calculé en fonction des coordonnées GPS obtenues lors de la géolocalisation.

```
mc.animateTo(centreInitial);
mc.setZoom(zoomInitial);
mapView.setSatellite(true);
mapView.setBuiltInZoomControls(true);
```

On utilise le contrôleur (`mc`) de l'objet `mapView` pour se déplacer (`animateTo()`) au centre choisi et appliquer la valeur de zoom (`setZoom()`) définie plus haut. On indique que l'on veut un affichage de type satellite (`setSatellite()`) ; la valeur de zoom est alors adaptée si Google possède des images satellites à la précision voulue (d'où l'importance de faire cela après le `setZoom()`). La méthode `setBuiltInZoomControls()` permet quant à elle de spécifier si on désire ou non avoir le contrôleur de zoom sur la carte.

```
Toast.makeText(this,
    "Lat : "+latitude+" Lng : "+longitude+" Ray : "+rayon,
    Toast.LENGTH_LONG).show();
```

Dernière pierre à cette activité : on fait apparaître un message indiquant les coordonnées du terminal et le rayon choisi au chargement de la carte. On convient qu'il s'agit plus là d'une preuve de bon fonctionnement que de la réponse à un besoin utilisateur criant.

4 EXÉCUTION DE L'APPLICATION

4.1 AndroidManifest.xml

Avant d'exécuter notre application, il nous faut encore régler deux ou trois paramètres. Ouvrez pour cela le fichier `AndroidManifest.xml` qui se trouve à la racine du projet. Ici aussi, il s'agit d'un fichier XML modifiable via une interface graphique.

Allez dans l'onglet ***Application >> Application Nodes >> Add... >> Uses Library >> OK***. Dans le champ ***Name***, insérez « com.google.android.maps ». Cliquez à nouveau sur ***Add... >> Activity >> OK***. Dans le champ ***Name****, insérez « MyMap ». Ces actions ajoutent la bibliothèque des cartes Google au projet et identifient la classe `MyMap` comme une activité du projet (la classe `MyFirstActivity` l'étant déjà depuis sa création).

Allez ensuite sur l'onglet ***Permissions >> Add... >> Uses Permission >> OK***. Dans le champ ***Name***, insérez « android.permission.ACCESS_FINE_LOCATION ». Cliquez à nouveau sur ***Add... >> Uses Permission >> OK***. Dans le champ ***Name***, insérez « android.permission.INTERNET ». Vous venez d'autoriser les accès à Internet et au GPS à votre application, deux choses dont elle a besoin pour fonctionner correctement.

N'oubliez surtout pas d'enregistrer le fichier.

4.2 Lancement de l'émulateur

On crée maintenant une nouvelle configuration d'exécution. Pour ce faire, cliquez sur ***Run >> Run Configurations... >> Clic droit sur Android Application >> New***. Dans le champ ***Name***, insérez « launch_MyFirstApp », dans ***Project***, sélectionnez ***MyFirstApp*** et dans ***Launch Action***, cochez ***Do Nothing***. Cliquez

Fig. 2 : Représentation graphique de la seconde activité (carte) de l'application

sur l'onglet ***Target***. Vous pouvez alors voir la liste des AVD utilisables pour l'émulation de votre application. Cochez ***my_avd***. Enfin, cliquez sur ***Apply*** puis sur ***Close***.

Vous pouvez maintenant émuler Android avec votre AVD en cliquant sur ***Run >> Run Configurations... >> launch_MyFirstApp >> Run***.

Une fois l'émulateur démarré, il vous faut simuler un fonctionnement du GPS en fournissant des coordonnées au terminal. Pour ce faire, ouvrez une console et entrez-y la commande `telnet 127.0.0.1 5554`. Normalement, vous voilà connecté au terminal Android émulé. Vous pouvez envoyer des coordonnées GPS à l'aide de la commande `geo fix 3.8266217708587646 44.71298250621202 978`. La page **[SIMU_GPS]** vous donnera plus d'explications sur le sujet (car ici, vous venez de signifier au terminal émulé qu'il se trouve près du château de Barres à Langogne, ce qui n'est pas forcément votre coin de France préféré). En attendant, vous pouvez exécuter votre programme. Il vous présentera le menu de la première activité, puis, lorsque vous aurez choisi un rayon, vous affichera un écran à l'image de la figure 2.

CONCLUSION

Nous nous arrêtons ici pour cette première partie. Certains restent sans doute sur leur faim, car notre application n'est pas des plus utiles au stade où elle en est. Ce point sera corrigé dans la partie 2. Nous traiterons alors de l'interrogation d'un service web afin de récupérer une liste de lieux. Nous verrons également comment placer ces lieux sur la carte et rendre l'ensemble interactif.

Notes

- **[DECHETTERIES]** Il fallait bien trouver un type de lieu remarquable... Or tout le monde sait ce qu'est une déchetterie, cela flattera l'écocitoyenneté du lecteur, et *last but not least*, on possède déjà l'adresse d'un service web fournissant une liste de ces établissements.
- **[ANDROID_1.6]** Tout au long de cette étude, nous utiliserons la version 1.6 d'Android. Certains se sentiront peut-être frustrés du fait que Google soit déjà passé aux versions 2.x et que nous ne soyons donc pas à la pointe. Maintenant, il est intéressant de noter que la 1.6 est certainement la version répandue sur le plus grand nombre de terminaux et que l'on trouve encore aujourd'hui des appareils en 1.5 (dont la célèbre Archos 5IT). En outre, bien qu'il puisse y avoir certaines différences entre les versions, l'objectif est ici de présenter les concepts de base qui, eux, changent plus lentement.
- **[BASE]** La complexité basique de notre programme de démonstration fait qu'il aurait probablement pu fonctionner sur la version 1.5 de la plate-forme.
- **[LAYOUT]** Comme en CSS, où l'on place les objets en position `absolute` ou `relative`, on trouve un `AbsoluteLayout`, un `RelativeLayout`, etc. Le `LinearLayout` à orientation verticale indique que tous les objets qu'il contient sont placés à la suite, les uns en dessous des autres.
- **[TEXTVIEW]** Dans tous les cas, la documentation fournie par Google décrit précisément tous les attributs. Pour l'objet TextView, c'est à cette adresse : http://developer.android.com/reference/android/widget/TextView.html

Références

- **[INSTALL_IDE]** *Android Developers, Installing the Android 1.6 SDK*, http://developer.android.com/sdk/1.6_r1/installing.html
- **[INTENT]** *Android Developers, Intent Specification*, http://developer.android.com/reference/android/content/Intent.html
- **[API_KEY]** *Google Code, Obtaining a Maps API Key*, http://code.google.com/intl/en/android/add-ons/google-apis/mapkey.html
- **[SPEC_MAPACT]** *Google Code, MapActivity Specification*, http://code.google.com/intl/en/android/add-ons/google-apis/reference/com/google/android/maps/MapActivity.html
- **[SIMU_GPS]** *Android Developers, Location and Maps*, http://developer.android.com/guide/topics/location/index.html

Auteur : Marc de Verdelhan des Molles

- mdeverdelhan@gmail.com
- http://www.verdelhan.eu/
- Ingénieur Informatique INSA Lyon

Ma première application Android - 2nde partie

Auteur

- Marc de Verdelhan des Molles

Dans la première partie de cette étude, le lecteur a pu appréhender certains concepts basiques du développement sous Android. Dans la réalisation de notre application, nous en étions restés à l'affichage d'une carte Google centrée sur la position géographique de l'utilisateur. Cette seconde partie arrive en complément de la précédente et porte sur l'interrogation d'un service web et le positionnement d'éléments sur la carte.

1 PRÉAMBULE

Dans le précédent article, nous avions vu comment démarrer le développement d'une application Android. Nous avions créé des *layouts* pour nos interfaces graphiques et nous avions expliqué comment passer d'une activité à l'autre. Certains lecteurs s'étaient sentis un peu lésés sachant qu'à part afficher une carte Google, notre programme ne faisait pas grand chose. Afin qu'il rende un service plus utile, on désire décorer la carte Google d'éléments sémantiquement intéressants pour l'utilisateur. C'est justement l'objet de cette seconde partie.

Comme nous l'avions indiqué le mois précédent, nous choisissons d'afficher les déchetteries se situant dans le rayon choisi, ainsi que la position de l'utilisateur. Afin de pouvoir les situer sur la carte, il faut auparavant obtenir la liste des centres de tri et de recyclage. Ceci se fait en interrogeant un service web.

2 DESCRIPTION DU SERVICE WEB

On part justement du principe que l'on possède un service web potentiellement interrogeable à l'aide d'une URL de type http://un-service-web.tld/find-decheteries.php?lat=XXX&lng=YYY&radius=ZZZ **[URL_WS]**.

Ce service retourne un flux XML dont voici un exemple :

```
<?xml version="1.0" encoding="utf-8"?>
<markers>
    <marker nom="DÉCHÈTERIE DE LANGOGNE"
       adresse="N 88 Route de Pignol ZI, 48300, LANGOGNE"
      lat="44.720565795898"
       lng="3.830185413361" />
    <marker nom="DECHETERIE DE CHATEAUNEUF DE RANDON"
      adresse="48000, CHATEAUNEUF DE RANDON"
      lat="44.640850067139"
       lng="3.674935102463" />
    <marker nom="DÉCHÈTERIE DE RIEUTORT DE RANDON"
       adresse="Route de Saint-Chély, 48700, RIEUTORT-DE-RANDON"
       lat="44.614322662354"
       lng="3.490288019180" />
</markers>
```

Comme vous pouvez le constater, la structure de ce flux est des plus simples. L'élément de base est le **marker**. Chaque **marker** correspond à une déchetterie et possède quatre attributs : le **nom**, l'**adresse**, la latitude (**lat**) et la longitude (**lng**). Le second élément est représenté par les balises **markers**. Il n'est rien d'autre qu'une liste d'éléments **marker**.

3 UTILISATION DE L'API SAX

On veut exécuter ce que les anglophones appellent le *parsing* du flux XML présenté précédemment. Pour cela, on utilisera l'API SAX **[CHOIX_SAX]**. Ainsi, on commence par créer une classe **Marker.java** censée représenter un élément **marker**.

```
public class Marker {

    /** Attributs de l'élément XML marker */
    private String nom;
    private String adresse;
    private String lat;
    private String lng;

    public Marker() { }

    /** @return the nom */
    public String getNom() {
        return nom;
    }

    /** @param nom the nom to set */
    public void setNom(String nom) {
        this.nom = nom;
    }

    /**
     * Ajouter ici les accesseurs (getXXX / setXXX)
     * des autres attributs (adresse, lat, lng).
     */

    /** @return une string représentant un marker */
    public String toString() {
        StringBuffer buf = new StringBuffer();
        buf.append("Marker: Nom='" + this.nom + "', ");
        buf.append("Adresse='" + this.adresse + "', ");
        buf.append("Lat='" + this.lat + "', ");
        buf.append("Lng='" + this.lng + "'");
        return buf.toString();
    }
}
```

Cette classe reprend juste chaque attribut d'un élément **marker** pour en faire un attribut de classe.

On crée ensuite la classe correspondant à l'élément **markers**. On l'appelle **ListeMarkers.java**.

```
public class ListeMarkers extends ItemizedOverlay<Marker> {

    private ArrayList<Marker>
                listeDesMarkers = new ArrayList<Marker>();

    private Context context;

    public ListeMarkers(Context context,
                        Drawable defaultMarker) {
        super(boundCenterBottom(defaultMarker));
        this.context = context;
        populate();
    }

    /**
     * Ajoute un marker à la liste
     * @param m le marker à ajouter
     */
    public void addMarker(Marker m) {
        listeDesMarkers.add(m);
        populate();
    }
   /** @return string représentant la liste des markers */
    public String toString() {
        /**
        * A implémenter comme vous voulez...
         */
    }

    @Override
    protected Marker createItem(int i) {
        return listeDesMarkers.get(i);
    }

    @Override
    public int size() {
        return listeDesMarkers.size();
    }

    @Override
    protected boolean onTap(int index) {
        return super.onTap(index);
    }
}
```

On précise que la classe **ListeMarkers** hérite d'**ItemizedOverlay<Marker>**. Concrètement, cela signifie qu'il s'agit d'une couche (**Overlay**) composée d'*items* (**Itemized**) de type Marker (**<Marker>**) à superposer à la carte Google afin que les markers en question soient représentés. Un tel héritage impose de redéfinir la méthode **onTap()** appelée en cas de clic (ou de frappe du doigt) sur un marker affiché sur la carte. Pour l'instant, on laisse cette méthode vide de toute action intéressante. On note enfin les deux paramètres (**context** et **defaultMarker**) passés au constructeur de cette classe. Ils seront utiles lorsque nous voudrons afficher les différents points sur la carte et interagir avec eux. Nous y reviendrons tout à l'heure.

Maintenant que la structure du flux XML est représentée à l'aide du modèle objet, on peut créer la classe gérant les événements SAX.

```
public class SAXEventHandler extends DefaultHandler {

    /** Liste des markers */
    private ListeMarkers listeDesMarkers;

    /** Pile utilisée pour le parsing */
    private Stack<Object> stack;

    private Context context;

    private Drawable defaultMarker;

    public SAXEventHandler(Context context,
                           Drawable defaultMarker) {
        super();
        this.stack = new Stack<Object>();
        this.context = context;
        this.defaultMarker = defaultMarker;
    }

    /** @return la liste listeDesMarkers */
    public ListeMarkers getListeDesMarkers() {
        return listeDesMarkers;
    }

    /**
    * Element de début trouvé.
     */
    public void startElement(String uri,
                             String localName,
                             String qName,
                            Attributes atts) {
        if(localName.equals("markers")) {
            // Mise dans la pile d'un élément <markers>
            stack.push(new ListeMarkers(context, defaultMarker));
        } else {
            if(localName.equals("marker")) {
                // Mise dans la pile d'un élément <marker>
                // (sans S final cette fois)
                Marker nMarker = new Marker(
```

```
                        atts.getValue(uri, "nom"),
                        atts.getValue(uri, "adresse"),
                        atts.getValue(uri, "lat"),
                        atts.getValue(uri, "lng"));
                stack.push(nMarker);
            } else {
                Log.d("SAX ECO", "ELEMENT INATTENDU");
            }
        }
    }

    /**
     * Element de fin trouvé.
     */
    public void endElement(String uri,
                           String localName,
                           String qName) {
        Object tmp = stack.pop();
        if( localName.equals("markers")) {
            this.listeDesMarkers = (ListeMarkers)tmp;
        } else {
            if(localName.equals("marker") ) {
                ((ListeMarkers)stack.peek()).addMarker((Marker)tmp);
            } else {
                Log.d("SAX ECO", "ELEMENT INATTENDU");
                stack.push(tmp);
            }
        }
    }
}
```

Cette dernière classe permet de construire la liste des markers, sous la forme d'un objet **ListeMarkers**, composée d'objets **Marker**. Pour ce faire, elle parcourt le flux XML et réagit aux événements (**EventHandler**) de type balise-de-début-trouvée (**startElement()**) et balise-de-fin-trouvée (**endElement()**).

Vous noterez que le constructeur de la classe prend en paramètres le contexte de l'application (**context**) et un objet **Drawable** (**defaultMarker**). Théoriquement, il est possible de se passer de ces paramètres pour la construction des objets à partir du flux. En revanche, ils sont nécessaires lors de l'appel au constructeur de la classe **ListeMarkers** (vue plus haut).

Cet article ne doit pas commettre l'erreur de trop s'attarder sur la méthode SAX. Cette dernière n'est pas l'objet de notre étude et est déjà suffisamment documentée sur Internet. C'est pourquoi nous inviterons le lecteur désirant plus d'explications sur le sujet à consulter le site du projet SAX **[PROJET_SAX]**. En attendant, notre architecture SAX est en place, nous pouvons consulter le service web.

4 CONSULTATION DU SERVICE WEB

A la classe **MyMap.java**, on ajoute une méthode privée dont la signature est la suivante : **private URL buildUrl(double latitude, double longitude, int rayon);**. C'est-à-dire qu'elle retourne un objet **URL** (correspondant à l'URL du service web à interroger) à partir des coordonnées GPS et du rayon de recherche.

On insère également la méthode faisant le parsing du fichier XML rendu par le service web.

```
/**
 * Retourne la liste des déchèteries après parsing
 * du web service
 * @param url URL à parser
 * @return une liste de markers tirée de l'URL
 */
private ListeMarkers parseXml(URL url) {
    // Création du parseur SAX
    SAXParserFactory spf = SAXParserFactory.newInstance();
    SAXParser sp = null;
    try {
        sp = spf.newSAXParser();
    } catch (Exception e) {
        e.printStackTrace();
    }

    // Récupération du lecteur XML depuis le parseur SAX
    // nouvellement créé
    XMLReader xr = null;
    try {
        xr = sp.getXMLReader();
    } catch (SAXException e) {
        e.printStackTrace();
    }

    /**
     * Création d'un intercepteur d'événements
     * et association de ce hander au XMLReader
     */
    SAXEventHandler saxEventHandler = new SAXEventHandler(
        this,
        this.getResources().getDrawable(R.drawable.marker_decheterie));
    xr.setContentHandler(saxEventHandler);
    // Début du parsing du contenu de l'URL
    InputSource in = new InputSource();
    try {
        // Récupération du contenu de l'URL
        in.setByteStream(url.openStream());
    } catch (IOException e) {
        e.printStackTrace();
    }
    try {
        xr.parse(in);
    } catch (Exception e) {
        e.printStackTrace();
    }
    // Fin du parsing
    return saxEventHandler.getListeDesMarkers();
}
```

Concrètement, on crée un objet **XMLReader** (**xr**) permettant de lire un flux XML. A l'aide de la méthode **setContentHandler()**, on lui précise l'intercepteur d'événements que l'on veut utiliser (cf. classe **SAXEventHandler** vue plus haut). On récupère le fichier XML dans un objet **InputSource** (**in.setByteStream(url.openStream());**). Enfin, on *parse* ce fichier avec **xr**. On termine la méthode en retournant la liste des markers sous la forme d'un objet **ListeMarkers**.

On notera également les paramètres donnés au constructeur de la classe **SAXEventHandler**. Ceux-ci correspondent au contexte (**Context**) et à un objet **Drawable** servant d'icône à chaque élément (marker/déchetterie) placé sur la carte. Ce symbole est identifié par **R.drawable.marker_decheterie**. Au préalable, on aura ajouté au projet (dans le dossier res/drawable/) un fichier **marker_decheterie.png** (25px*25px) pour que le programme sache où trouver la ressource graphique à afficher.

Bien sûr, pour que tout cela s'exécute, il faut qu'à la fin de la méthode **onCreate()** (de la classe **MyMap**), on ajoute les lignes suivantes :

```
URL url = buildUrl(latitude, longitude, rayon);
ListeMarkers listeDesMarkers = parseXml(url);
mapView.getOverlays().add(listeDesMarkers);
```

La dernière de ces trois lignes ajoute la couche des items (marker) à la liste des couches de la `mapView` correspondant à la carte Google. Comme vous pouvez vous en douter, il est possible d'en ajouter autant que l'on veut et ainsi d'enrichir la carte avec de nouveaux éléments.

5 INTERACTION AVEC LES ITEMS

Un des derniers points de notre cahier des charges consiste à afficher une boîte de dialogue lorsque l'utilisateur clique (ou frappe du doigt) sur une déchetterie. On veut quelque chose de simple, c'est-à-dire une fenêtre affichant uniquement le nom et l'adresse de cette déchetterie. Pour cela, on a besoin d'un layout spécifique à la boîte de dialogue. On crée donc le fichier `dialog.xml` dans le dossier des layouts et on y copie le code suivant :

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content">
    <TextView android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:id="@+id/nom"
        android:textStyle="bold"
        android:layout_alignParentLeft="true"
        android:textColor="#fff"
        android:textSize="13sp">
    </TextView>
    <TextView android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:id="@+id/adresse"
        android:layout_below="@+id/nom"
        android:textSize="11sp"
        android:layout_marginBottom="8sp">
    </TextView>
</RelativeLayout>
```

Rien de bien sorcier par rapport à ce qui a déjà été vu dans la partie précédente. On ne fait ici que définir un positionnement pour le nom et l'adresse au sein de la boîte de dialogue.

Revenons maintenant à la classe `ListeMarkers.java` et plus particulièrement à sa méthode `onTap()`. Nous l'avions laissée vide, or il nous faut en implémenter le contenu car elle correspond à ce qui doit être exécuté lors du clic sur la couche (`Overlay`) des déchetteries. Ainsi, avant l'instruction `return`, on ajoute :

```
// Création de la boite de dialogue
final Dialog dialog = new Dialog(context);
dialog.requestWindowFeature(Window.FEATURE_NO_TITLE);
dialog.setContentView(R.layout.dialog);
dialog.setCancelable(true);
dialog.setCanceledOnTouchOutside(true);

// Affichage du nom et de l'adresse
TextView viewNom = (TextView) dialog.findViewById(R.id.nom);
viewNom.setText(listeDesMarkers.get(index).getNom());
TextView viewAdresse
          = (TextView) dialog.findViewById(R.id.adresse);
viewAdresse.setText(listeDesMarkers.get(index).getAdresse());

dialog.show();
```

Ici, on commence par créer une boîte de dialogue. Il nous faut pour cela lui passer le contexte lors de sa construction. On comprend mieux maintenant pourquoi il était nécessaire de passer ce dernier en paramètre. On précise que la boîte de dialogue ne doit pas avoir de zone de titre (`FEATURE_NO_TITLE`), que ses éléments sont agencés suivant le layout `dialog` créé précédemment et qu'elle peut être fermée à l'aide du bouton retour (`setCancelable(true)`) ou lors d'un clic à l'extérieur de sa zone graphique (`setCanceledOnTouchOutside(true)`).

On associe les attributs (`nom`, `adresse`) de l'item sur lequel on a cliqué (`listeDesMarkers.get(index)`) aux TextView du layout que l'on vient de fournir à la boîte de dialogue.

Enfin, on affiche la boîte de dialogue avec la méthode `show()`.

6 DERNIERS DÉTAILS

Une bonne application est composée de détails. Certains sont triviaux à implémenter, d'autres beaucoup plus complexes. Quoi qu'il en soit, si votre programme ne possède pas ce petit plus qui fera que l'utilisateur le préférera à un concurrent, vous pouvez le jeter. Nous n'irons pas jusqu'à dire que l'exemple déroulé jusqu'alors constitue une bonne application. Néanmoins, voici deux détails graphiques qu'il me semble vital de travailler.

6.1 L'icône du programme

L'icône du programme est tout simplement l'icône sur laquelle l'utilisateur clique pour lancer l'application. Lors de la création d'un nouveau projet Android, il en existe une par défaut. Vous pouvez la modifier en remplaçant le fichier `res/drawable/icon.png` par une image de votre choix. La seule contrainte étant qu'elle doit mesurer 48px*48px.

6.2 Le marker de position du terminal

Le marker de position du terminal n'est ni plus ni moins que l'icône placée sur la carte indiquant la position géographique du terminal Android (et donc souvent de l'utilisateur). Il existe plusieurs méthodes pour l'afficher. Ceci étant, à mon sens, celle qui suit est la plus simple et celle qui offre le plus de perspectives d'évolutions.

On ajoute d'abord le fichier `marker_position.png` dans le dossier res/drawable/. On crée la classe `YouAreHereOverlay.java` servant de couche (`Overlay`) pour l'icône.

```
public class YouAreHereOverlay
                    extends ItemizedOverlay<OverlayItem> {

    private OverlayItem youAreHereItem;

    public YouAreHereOverlay(Drawable defaultMarker) {
        super(boundCenterBottom(defaultMarker));
    }

    public void setYouAreHereItem(OverlayItem youAreHereItem) {
        this.youAreHereItem = youAreHereItem;
        populate();
    }

   @Override
   protected OverlayItem createItem(int i) {
        return youAreHereItem;
    }
   @Override
    public int size() {
        return 1;
    }
}
```

Rien de révolutionnaire : il s'agit de la même classe (simplifiée tout de même) que pour la liste des déchetteries. On l'utilise à la fin de la méthode `onCreate()` de la classe `MyMap` en y ajoutant :

```
YouAreHereOverlay youAreHereOverlay = new YouAreHereOverlay(
    this.getResources().getDrawable(R.drawable.marker_position));
youAreHereOverlay.setYouAreHereItem(new Marker(
                                        "You are here!",
                                        "Vous êtes ici!",
                                        latitude+"",
                                        longitude+""));
mapView.getOverlays().add(youAreHereOverlay);
```

A l'origine, la classe `Marker` correspond à un emplacement de déchetterie. On la recycle en s'en servant pour le marker de position du terminal. Enfin, on superpose cette autre couche à la carte Google.

CONCLUSION

Dans cette seconde partie de notre introduction au SDK Android, nous avons vu comment placer des éléments sur une carte Google. Auparavant, nous avions indiqué le moyen d'interroger un service web afin d'en récupérer les données. Enfin, nous avons montré les mécanismes d'interaction avec les *items* de la carte. Un exemple de résultat vous est montré à travers les figures 1 et 2.

Si vous êtes arrivé jusque là, il me faut vous féliciter. Vous n'avez pas décroché, vous vous êtes battu, vous n'avez jamais songé à renoncer ! Lorsque l'on débute sur une plate-forme, il n'est pas trivial de se lancer directement dans un programme si complexe. On préfère souvent le *Hello World !* minimaliste. Quoi qu'il en soit, ces deux articles vous ont normalement permis de créer une application Android fonctionnelle, utile, soignée d'un point de vue graphique (les boutons de différentes couleurs : c'est quelque chose, hein ?), en un mot : parfaite. On sent maintenant beaucoup de lecteurs désireux de récupérer la source pour l'améliorer et en faire profiter les autres. A tous ceux-là : sachez que ce dernier rêve peut se réaliser ! Pour cela, il vous suffit de télécharger le code à cette adresse : **[SOURCE]**. Bonne route !

Fig. 1 : Markers affichés sur la carte

Fig. 2 : Boîte de dialogue affichée lors d'un clic sur un item

Notes

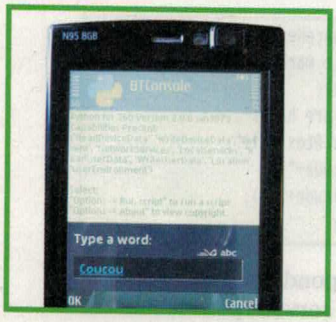
- **[URL_WS]** Où XXX correspond à la latitude de la position de l'utilisateur, YYY à sa longitude et ZZZ au rayon de recherche.
- **[CHOIX_SAX]** On aurait pu traiter le flux plutôt à l'aide de la méthode DOM, d'autant plus qu'aucune contrainte technique n'imposait SAX. Ceci étant, il fallait bien faire un choix. Pour plus de renseignements sur les différences existant entre ces deux approches, je vous invite à vous rendre à la page **[DIFF_SAX_DOM]**.

Auteur : Marc de Verdelhan des Molles

Références

- **[DIFF_SAX_DOM]** Developpez.com, FAQ Java XML : Que sont les API SAX, DOM et XSLT ?, http://java.developpez.com/faq/xml/?page=generalitesXML#saxDomXslt
- **[PROJET_SAX]** Saxproject.org, *Quickstart with SAX*, http://www.saxproject.org/quickstart.html
- **[SOURCE]** Marc de Verdelhan des Molles, *AndDeche!*, http://www.verdelhan.eu/wiki/doku.php?id=anddeche
- **[ANDDEV]** Anddev.org, *Android Development Community | Android Tutorials*, http://www.anddev.org/
- **[DEV_FRANDROID]** FrAndroid, *FrAndroid Développement*, http://dev.frandroid.com/

Développez des applications



Le développement d'applications pour smartphones ou, plus largement, pour téléphones mobiles, est généralement une affaire de SDK spécialisés et de langages compilés comme C ou C++. Depuis quelque temps cependant, nos « mobiles » sont devenus suffisamment puissants pour supporter un interpréteur. La création d'applications s'en trouve grandement simplifiée.

Auteur

- Denis Bodor

Notre plate-forme de référence sera ici un mobile de marque Nokia fonctionnant sous Symbian S60. Ceci s'avère être une large gamme de modèles et nos applications pourront très simplement être portées d'un périphérique à un autre. Seul impératif, il sera nécessaire de disposer d'un interpréteur Python. Dans le cadre de cet article, le téléphone utilisé est un Nokia N95 8GB. Celui-ci se différencie du modèle N95 précédent par le fait qu'il intègre un support de stockage flash de 8 Go interne en lieu et place d'un lecteur de microSD. Le N95 8GB fonctionne sous Symbian « S60 3rd Ed FP1 » (S60 3ème édition Features Pack 1). La désignation exacte du système présent sur votre mobile Nokia est importante en ce qui concerne le choix de l'environnement Python à télécharger et installer. Voyez http://www.forum.nokia.com/devices/matrix_s60_1.html pour la liste des mobiles de la marque et leur version respective de Symbian.

1 L'ENVIRONNEMENT PYTHON

On ne présente normalement plus Python, un langage de programmation interprété orienté objet reposant sur un typage dynamique fort, une gestion automatique de la mémoire (ramasse-miettes) et un système de gestion d'exceptions. Python est adapté pour une vaste collection de développements allant des outils systèmes en passant par les serveurs ou encore les jeux ou les interfaces graphiques.

De manière surprenante, Python est un langage interprété, il s'avère également très efficace dans le monde de l'embarqué où il se fait, petit à petit, une place de choix là où Perl, Ruby ou Tcl n'ont finalement pas vraiment percé. Ainsi, le développeur de/pour systèmes embarqués fait maintenant connaissance avec un langage permettant un prototypage rapide des applications.

2 PYTHON FOR S60

Le portage de Python sur plate-forme Symbian S60 est en grande partie le travail des équipes Nokia. Le constructeur finlandais n'en est pas à ses premiers pas dans le monde de l'open source et c'est un peu sans surprise qu'on constate que l'ensemble du code est produit sous une licence de logiciel libre. Le 11 février dernier, Nokia, via son site d'hébergement (garage.maemo.org), annonçait une nouvelle version 2.0.0 de son *Python for S60* (alias PyS60) répondant aux caractéristiques suivantes (liste non exhaustive) :

- code source entièrement diffusé et en open source ;
- support de l'envoi, la réception et l'analyse de SMS ;
- accès étendu à l'API audio (enregistrement et playback) ;
- fonctions graphiques 2D, Images, et applications plein écran ;
- accès à l'API de l'appareil photo intégré et des fonctions de capture d'écran ;
- gestion des contacts et de l'organiseur ;
- accès aux informations systèmes (IMEI, espace disque, mémoire, etc.) ;
- affichage de texte formaté (polices, styles couleurs) ;
- prise en charge des événements en provenance du clavier ;
- numérotation téléphonique ;
- accès aux fonctions réseau GPRS et Bluetooth ;
- console Python locale et distante ;
- support des widgets natif.

pour Symbian en Python

Une certaine partie de ces fonctionnalités sont toutes récentes et sont arrivées avec la version 2.0.0. C'est le cas, par exemple, des fonctions de réception de SMS. D'autres fonctionnalités sont présentes de longue date, comme la récupération des informations systèmes. Il faut également prendre en considération le fait qu'une partie des fonctions ne sont pas disponibles sur tous les modèles de téléphones. La prise en charge du *touchscreen*, de l'accéléromètre ou du GPS interne dépendra, tout naturellement, du matériel en présence. Sachez cependant que les trois éléments qui viennent d'être cités sont pris en charge et pilotables (du moins en partie) depuis un code Python.

L'installation de l'environnement Python pour votre mobile ainsi que les outils nécessaires pour le développement côté PC nécessitent le téléchargement d'une archive (`PythonForS60_2.0.0.tar.gz`) formant le *Ensymble developer utilities for Symbian OS*. Celui-ci comprend :

- une documentation PDF conséquente pour développeurs sous le nom « *S60 Module Reference* » ;
- les différents éléments à installer sur votre mobile (`PyS60Dependencies/`) sous la forme de fichiers/paquets SIS ;
- l'outil `ensymble.py` permettant de créer, manipuler et signer des paquets SIS.

Notez que ce kit de développement est normalement conçu pour S60 3rd Ed ou, en d'autres termes, pour Symbian OS v9.1 et supérieur. Les premières et secondes éditions sont supportées via l'ajout d'un programme spécifique.

3 UTILISATION DE LA CONSOLE PYTHON EN BLUETOOTH

Développer en Python signifie généralement, en particulier sur une plate-forme embarquée, prototypage rapide. Les manipulations : édition de code, mise en paquet SIS, installation, exécution, ne sont pas ce qu'on peut appeler quelque chose d'efficace en termes de développement rapide. Il existe une bien meilleure solution consistant à utiliser un PC sous GNU/Linux équipé d'un dongle BlueTooth, pour établir un lien RFCOMM directement vers l'environnement Python du mobile.

On commence donc, côté PC, par un reset de l'interface BlueTooth :

```
% sudo hciconfig reset
hci0:   Type: USB
        BD Address: 00:0A:9A:00:08:8C ACL MTU: 339:4 SCO MTU: 0:0
        UP RUNNING PSCAN ISCAN
        RX bytes:1002 acl:0 sco:0 events:25 errors:0
        TX bytes:338 acl:0 sco:0 commands:24 errors:0
```

On crée/enregistre ensuite un port série sur le canal 2. Comme le précise la documentation sur le Wiki Nokia, les canaux 1 et 3 risquent de poser problème *for some reason* (sic) :

```
% sudo sdptool add --channel=2 SP
Serial Port service registered
```

Enfin, on déclenche l'attente de connexion :

```
% rfcomm listen rfcomm2 2
Waiting for connection on channel 2
```

Il faut ensuite se tourner vers le mobile et lancer le Shell Python. Dans le menu des options, on choisira de lancer la console BlueTooth. Si votre PC n'est pas encore connu du mobile, la procédure d'association classique sera lancée. De la même manière, si le BlueTooth n'est pas activé, la mise en service vous sera proposée. En cas de problème, jetez un œil à votre `/etc/bluetooth/hcid.conf`. Dès lors, la connexion est établie et la console sur le mobile affiche l'adresse de la machine distante et l'état de la connexion. Côté Linux, `rfcomm` fera de même :

```
Connection from 00:22:FD:3C:B8:AA to /dev/rfcomm2
Press CTRL-C for hangup
```

Il ne vous reste plus ensuite qu'à utiliser votre émulateur de terminal série préféré (ici `screen`) pour vous connecter au mobile (`screen /dev/rfcomm2`). Vous devrez obtenir l'invite classique de Python et pouvez dès à présent saisir vos instructions avec tous les avantages que cela suppose (clavier, copier/coller, log, etc) :

```
>>> print "coucou"
coucou
>>> import e32
>>> e32.pys60_version
'2.0.0 svn3873'
```

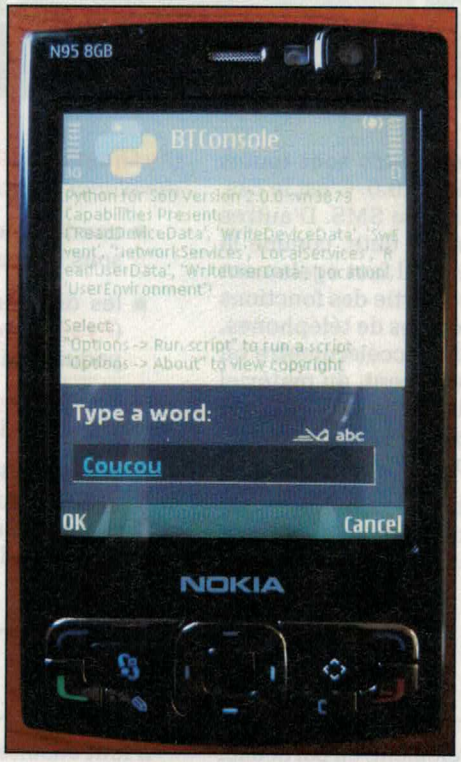
Bien entendu, ceci fonctionnera aussi bien pour les essais en mode texte que pour les applications et le code reposant sur l'interface graphique Symbian (GUI).

4 QUELQUES EXEMPLES SIMPLES

Nous venons de le voir implicitement, il est parfaitement possible de créer des applications en mode texte. Bien entendu, ce n'est pas avec des `print "Bonjour Monde"` que nous pourrons construire une application satisfaisante. Le *binding* Python créé par Nokia offre bien plus que cela. Commençons avec un simple exemple :

```
>>> import appuifw
>>> data = appuifw.query(u"Type a word:", "text")
>>> print data
Coucou
>>>
```

appuifw est le module permettant l'accès à la plupart des fonctionnalités en rapport avec la GUI Nokia. **query** nous permet d'afficher une demande à l'utilisateur sous la forme d'une boîte de dialogue avec un champ de saisie unique. Dans le cas présent, ce champ est de type texte. Nous aurions tout aussi bien pu spécifier un champ numérique (**"number"**), une date (**"date"**), une heure (**"time"**), une question oui/non (**"query"**) ou encore un mot de passe (**"code"**). Pour afficher simplement un message, c'est la fonction **note** qui pourra être utilisée.



De la même manière, vous pouvez utiliser des menus popups, des listes sélectionnables, des boîtes de dialogue à double champ, etc. Toutes ces fonctions sont largement documentées dans les quelques 290 pages du « S60 Module Reference ».

La console Bluetooth ou, pour les plus courageux, la console interactive proposée par le Python ScriptShell, est très pratique pour évaluer des fonctions et faire quelques tests. Lorsqu'il s'agit de se pencher sur la réalisation d'une véritable application, il devient nécessaire de passer par une méthode plus complète.

Considérons le script suivant, que nous commenterons point par point :

```
import appuifw, e32, key_codes, sysinfo, graphics

def quit():
  # stops the scheduler
  app_lock.signal()

def hello():
  data = appuifw.query(u"Votre nom", "text")
  appuifw.note(u"Hello "+str(data)+",\nbienvenue dans Python.", "info")

def dabat():
  appuifw.note(u"Bat. : "+str(sysinfo.battery())+"%", "info")
```

Après importation de différents modules, nous définissons les fonctions de callbacks dont nous aurons besoin pour différentes interactions avec l'utilisateur. Nous avons ici un exemple des informations systèmes qu'il est possible de récupérer très facilement, **sysinfo.battery** pour le niveau d'énergie restant dans la batterie.

```
appuifw.app.title = u"Essai PyS60"
appuifw.app.screen = "normal"
```

Ces deux lignes permettent de définir l'aspect général de l'application, à commencer par son « titre » et le modèle qu'il faut utiliser. Ici, **normal** spécifie un écran avec barre de titre et étiquettes des touches en bas. D'autres options existent : **large** (sans barre de titre) et **full** (plein écran).

```
canvas=appuifw.Canvas()
w,h = canvas.size
img = graphics.Image.new((w,h))
img.clear(0x0000ff)
appuifw.app.body = canvas
```

Ceci est le corps de l'écran de l'application. Plusieurs options sont possibles et nous choisissons ici un Canvas permettant de dessiner en mode point sur l'écran. Mais l'avantage de ce modèle est la prise en charge simplifiée des touches du téléphone. Après avoir créé l'objet, nous récupérons ses dimensions (**w,h**) pour créer une image de taille correspondante (**img**) et la remplir de bleu. Il ne nous reste plus ensuite qu'à désigner le Canvas comme étant le corps de notre application graphique.

```
appuifw.app.menu = [(u"Hello", hello),(u"Batterie", dabat)]
canvas.bind(key_codes.EKey2, hello)
canvas.bind(key_codes.EKey3, dabat)
appuifw.app.exit_key_handler = quit
```

Voici la gestion de l'interactivité avec l'utilisateur. Nous avons ici à la fois la mise en place d'un menu à deux entrées respectivement rattachées aux fonctions callbacks précédemment déclarées. La fonction **canvas.bind** nous permet de faire de même pour les touches [2] et [3]. Enfin, nous spécifions le callback appelé lorsque l'application est quittée.

```
# create an instance of the active object
app_lock = e32.Ao_lock()
# starts a scheduler -> the script processes events (e.g. from the UI)
until lock.signal() is callled.
app_lock.wait()
```

Nous avons ici un exemple d'utilisation de la technologie *active object*, largement employée pour le développement d'applications Nokia. Celle-ci est utilisée pour implémenter un ordonnancement non préemptif à l'intérieur des *threads* systèmes. Une fois notre application entièrement configurée, nous créons une instance **Ao_lock** (*active object based synchronization service*). Nous démarrons ensuite l'ordonnanceur pour que notre code puisse gérer les événements tout en permettant aux autres actives objects du système d'être servis. Notre interface ne « gèle » donc pas et nos callbacks sont parfaitement exécutés dans le contexte du thread ayant lancé le **wait**.

Pour « relâcher » l'attente (**wait**), il nous suffit d'appeler **app_lock.signal()**. Chose que nous faisons dans notre fonction **quit()**.

5 QUELQUES EXEMPLES PLUS SPÉCIFIQUES

L'API S60 et les fonctionnalités à disposition via le binding Python sont très importantes. Il serait vain de toutes les lister ici et je vous invite à consulter le PDF de référence de Nokia à ce propos. Voici cependant quelques exemples intéressants en termes de découverte :

- Envoi de SMS

```
import appuifw messaging

nom = appuifw.query(u"Entrez un nom", "text")

txt = u"Coucou de " +nom
tel = "06nnnnnnnn"

messaging.sms_send(tel, txt)
```

- Réception de SMS

```
import inbox, appuifw, e32

def message_received(msg_id):
    box = inbox.Inbox()
    sms_text = box.content(msg_id)
    appuifw.note(u"contenu: " + sms_text , "info")
    app_lock.signal()

box = inbox.Inbox()
box.bind(message_received)
print "En attente de SMS..."
app_lock = e32.Ao_lock()
app_lock.wait()
print "Message traité."
```

- Envoi de MMS

```
import appuifw messaging

nbr = "06nnnnnnn"
txt = u"Et voici une image"

messaging.mms_send(nbr, txt, attachment='e:\\Images\\image1.jpg')
```

- Synthèse de son/paroles

```
import appuifw audio

text = appuifw.query(u"dire :", "text")
audio.say(text)
```

- Information sur la téléphonie

```
import appuifw e32 location

(mcc, mnc, lac, cellid) = location.gsm_location()
print u"MCC: " + unicode(mcc)
print u"MNC: " + unicode(mnc)
print u"LAC: " + unicode(lac)
print u"Cell id: " + unicode(cellid)

appuifw.note(u"Cell id: " + unicode(cellid), "info")
```

6 CONSTRUIRE UN PAQUET

Ici, rien de bien complexe. Le script **ensymble.py** permet de très facilement produire un paquet SIS à partir d'un simple script Python :

```
% ./ensymble.py py2sis helloworld.py --appname="Essai PyS60"
ensymble.py: warning: no application version given, using 1.0.0
ensymble.py: warning: no UID given, using auto-generated test-range UID 0xed009e31
ensymble.py: warning: no certificate given, using insecure built-in one
```

La notion de certification et de signature d'applications est très présente dans le monde de la téléphonie mobile. Lors de l'installation de ce SIS (**helloworld_v1_0_0.sis**) sur votre mobile, vous aurez à confirmer le fait que vous prenez la responsabilité de l'action. L'application n'est pas signée et de ce fait, Nokia dégage toute responsabilité. De la même manière, plusieurs messages d'avertissement et de confirmation s'afficheront lors de l'installation en vous informant, par exemple, des fonctions utilisées par l'application en question.

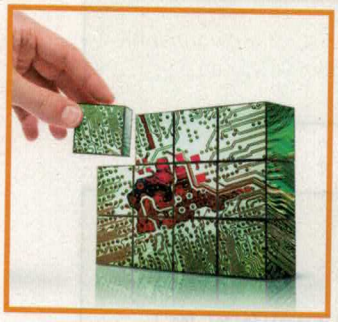
Nous ne présentons ici que l'aspect le plus simple de la confection de paquets SIS. Celle vous permettant de très rapidement produire et tester votre code. Sachez cependant qu'il existe une vaste collection d'options et de possibilités. Consultez la documentation fournie pour plus d'informations.

CONCLUSION

Comme vous venez de le voir, produire et tester du code Python pour un périphérique S60 3ème édition est quelque chose de très facile. Le travail réalisé par les développeurs de Nokia est très appréciable et il doit être salué comme il convient. Bon nombre de codes exemples fleurissent déjà un peu partout sur le Web. Nokia tente bien entendu de les référencer sur son Wiki. Pour en apprendre plus, c'est donc là une source non négligeable de documentation, tout comme les quelques centaines de pages du PDF de référence de l'API Python.

Auteur : Denis Bodor

Introduction à Buildroot



Le choix d'une distribution Linux embarquée n'est pas une tâche aisée. L'utilisateur a le choix entre la « réduction » d'une distribution classique, l'acquisition d'un produit commercial ou la construction d'une distribution from scratch. Une autre solution est envisageable, en l'occurrence, l'utilisation d'un outil de développement dédié. Il en existe plusieurs, citons LTIB, OpenEmbedded, PTXdist, Openwrt ou Buildroot. La facilité de prise en main, la puissance et la complexité sont très variables suivant les produits.

Auteur

- Pierre Ficheux

Le projet qui nous intéresse aujourd'hui a longtemps été d'un abord assez difficile car c'était plus un démonstrateur interne du projet uClibc qu'un véritable produit. Les choses ont bien changé depuis début 2009, et de ce fait, Buildroot est peut-être l'outil qu'il vous faut !

Dans cette introduction, nous rappellerons rapidement les différents éléments d'une distribution embarquée, puis nous décrirons les différentes phases d'utilisation de Buildroot. Afin de permettre au plus grand nombre de mettre en œuvre les exemples décrits, nous baserons nos tests sur l'émulateur QEMU (version ARM9). Nous en profiterons pour évoquer les possibilités de QEMU en tant qu'outil de test industriel.

Dans un deuxième article « Buildroot avancé », nous traiterons de concepts plus complexes, comme l'ajout du support d'une nouvelle carte mère ARM9 équipée du *bootloader* U-Boot ou l'utilisation d'une chaîne de compilation externe basée sur Glibc.

Les exemples présentés sont disponibles sur **http://pficheux.free.fr/articles/lmf/buildroot/exemples_intro.tgz**.

1 LES ÉLÉMENTS D'UNE DISTRIBUTION LINUX MINIMALE

La distribution Linux est constituée de deux éléments : le noyau et le *root-filesystem*. Dans un premier temps, nous créerons volontairement notre distribution « à la main », ce qui est somme toute assez simple.

Pour cela, il faut :

- un compilateur croisé ;
- un noyau Linux adapté à l'architecture (dans notre cas, l'émulateur QEMU/ARM9) ;
- Busybox pour la partie commandes en espace utilisateur.

1.1 Le compilateur croisé

Bien évidemment, il convient d'utiliser un compilateur croisé pour produire le noyau et les composants du root-filesystem. Nous avons simplifié la tâche en utilisant le compilateur ELDK-4.1 de DENX Software, version ARM. Une fois l'image ISO récupérée (voir l'adresse dans la bibliographie), l'installation est triviale.

```
$ sudo mount -o loop arm-2007-01-21.iso /mnt/tmp
$ /mnt/tmp/install -d /opt/ELDK41/arm
```

Une fois l'installation terminée, le compilateur sera disponible en ajoutant le chemin d'accès correct à la variable PATH. Il faut noter que cette version est basée sur la bibliothèque Glibc. Il existe une version équivalente basée sur la bibliothèque uClibc.

```
$ export PATH=/opt/sdk/ELDK41/arm/usr/bin:$PATH
$ arm-linux-gcc -v
Reading specs from /opt/sdk/ELDK41/arm/usr/bin/../lib/gcc/arm-linux/4.0.0/specs
Target: arm-linux
Configured with: /opt/eldk/build/arm-2007-01-21/work/usr/src/denx/
BUILD/crosstool-0.35/build/gcc-4.0.0-glibc-2.3.5-eldk/arm-linux/
gcc-4.0.0/configure --target=arm-linux --host=i686-host_pc-linux-gnu
--prefix=/var/tmp/eldk.bMi2nK/usr/crosstool/gcc-4.0.0-glibc-2.3.5-eldk/
arm-linux --with-headers=/var/tmp/eldk.bMi2nK/usr/crosstool/gcc-4.0.0-
glibc-2.3.5-eldk/arm-linux/arm-linux/include --with-local-prefix=/var/
tmp/eldk.bMi2nK/usr/crosstool/gcc-4.0.0-glibc-2.3.5-eldk/arm-linux/
arm-linux --disable-nls --enable-threads=posix --enable-symvers=gnu
--enable-languages=c,c++ --enable-shared --enable-c99 --enable-long-long
--enable-__cxa_atexit
Thread model: posix
gcc version 4.0.0 (DENX ELDK 4.1 4.0.0)
```

La compilation croisée du noyau et de Busybox nécessite de positionner les variables d'environnement **ARCH** et **CROSS_COMPILE**. Dans le cas présent, le plus simple est de créer un script **set_env_arm.sh** pour cela.

```
#!/bin/sh
PATH=/opt/sdk/ELDK41/arm/usr/bin:$PATH
ARCH=arm
CROSS_COMPILE=arm-linux-
export PATH ARCH CROSS_COMPILE
```

En exécutant ce script dans l'environnement courant, on peut alors utiliser le compilateur croisé pour Busybox et le noyau. Pour cela, on fera :

```
$ source ./set_env_arm.sh
```

ou :

```
$ . ./set_env_arm.sh
```

1.2 Le noyau et le root-filesystem

Le noyau à utiliser peut être choisi dans la dernière série des noyaux 2.6 puisque cette carte ne nécessite pas d'adaptation particulière. Dans le cas présent, nous avons utilisé un noyau 2.6.26. Concernant le root-filesystem, nous avons choisi d'utiliser la fonctionnalité **INITRD** du noyau Linux.

Le principal avantage est la mise en place très rapide dans le cas d'un test, car le système entier tourne en mémoire RAM.

Pour mettre en place le système, il faut :

1. Créer l'arborescence du root-filesystem dans un répertoire d'accueil.
2. Créer une image CPIO de ce répertoire. Pour cela, on utilise la commande suivante dans le répertoire du root-filesystem. L'option **-H newc** est très importante pour le bon fonctionnement du système. L'image **INITRD** à utiliser sera la version compressée de l'archive créée.

```
$ find . | cpio -o -H newc | gzip > /tmp/rootfs_arm.gz
```

3. Configurer et compiler le noyau avec **arm-linux-gcc**, l'image à utiliser sera le fichier **arch/arm/boot/zImage**.

Le root-filesystem étant basé sur Busybox, nous avons tout d'abord compilé une version récente de Busybox chargée sur http://www.busybox.net. Les variables **ARCH** et **CROSS_COMPILE** étant positionnées, il suffit d'exécuter les commandes suivantes :

```
$ make menuconfig
$ make
$ make CONFIG_PREFIX=<path_rootfs_arm> install
```

Si **<path_rootfs_arm>** est le répertoire contenant le root-filesystem, on obtient alors les fichiers suivants :

```
$ ls -l
total 12
drwxr-xr-x 2 pierre users 4096 févr. 15 18:43 bin
drwxr-xr-x 2 pierre users 4096 févr. 15 18:43 sbin
drwxr-xr-x 4 pierre users 4096 févr. 15 18:43 usr
```

Pour l'utilisation d'une image **INITRD**, il est indispensable de mettre en place un lien symbolique comme suit, car le noyau ira chercher par défaut le programme **/init** et non pas **/sbin/init**.

```
$ ln -s bin/busybox init
$ ls -l
total 12
drwxr-xr-x 2 pierre users 4096 févr. 15 18:43 bin
lrwxrwxrwx 1 pierre users   11 févr. 15 18:44 init -> bin/busybox
drwxr-xr-x 2 pierre users 4096 févr. 15 18:43 sbin
drwxr-xr-x 4 pierre users 4096 févr. 15 18:43 usr
```

Dans un deuxième temps, il faut mettre en place le répertoire **dev** contenant les fichiers spéciaux. Il existe des méthodes avancées pour cela, en se basant sur l'utilitaire **genext2fs**. Ce programme permet de créer les entrées dans **dev** sans avoir d'accès super-utilisateur et en utilisant une table décrivant précisément les fichiers à créer (cas de Buildroot). Dans notre cas, nous allons faire simple en utilisant la commande **MAKEDEV** de la distribution.

```
$ mkdir dev
$ sudo MAKEDEV -v -d dev generic console
```

ATTENTION, la syntaxe est différente sur Debian :

```
$ cd dev
$ sudo MAKEDEV -v generic console
```

Il reste à mettre en place les bibliothèques partagées nécessaires à la commande **busybox**. Pour cela, on peut utiliser le script **mklibs**. On passe à ce script la liste des programmes exécutables de la distribution (pour nous, **/bin/busybox**) ainsi que le chemin d'accès aux bibliothèques croisées (option **-L**).

```
$mkdir lib
$ mklibs -v --target arm-linux -D -L /opt/sdk/ELDK41/arm/arm/lib -d lib
bin/busybox
```

Le système est d'ores et déjà utilisable pour une première démonstration. On peut alors créer l'archive **CPIO** pour **INITRD**.

```
$find . | cpio -o -H newc | gzip > /tmp/rootfs_arm.gz
```

Pour configurer puis compiler le noyau, on utilise une séquence similaire si **<kernel_sources>** est le répertoire contenant les sources du noyau Linux.

```
$ cd <kernel_sources>
$ make menuconfig
$ make
```

Si l'on dispose déjà d'un fichier de configuration pour le noyau, on peut utiliser la séquence :

```
$ cd <kernel_sources>
$ cp <config_kernel_file> .config
$ make oldconfig
$ make
```

Finalement, nous obtenons les deux éléments de notre distribution de test :

- un noyau Linux statique (fichier **zImage**), destiné à une architecture matérielle donnée ;
- une image **INITRD** du root-filesystem (fichier **rootfs_arm.gz**).

Il reste à choisir l'outil de test permettant de valider notre distribution minimale.

2 L'OUTIL DE TEST QEMU

Il existe de nombreux articles sur Internet décrivant comment utiliser QEMU dans un environnement de développement classique (x86). La majeure partie des documentations décrivent l'exécution d'un autre système d'exploitation dans une session Linux exécutant QEMU. Pour cette raison, nous ne ferons pas de rappel sur les généralités concernant QEMU, le lecteur pourra se référer à la bibliographie en fin d'article et les nombreux pointeurs sur Internet.

Nous allons rapidement décrire comment utiliser QEMU comme outil de développement d'applications embarquées sur des architectures industrielles.

2.1 L'intérêt de QEMU pour le développement embarqué

Le principal défaut du développement industriel est la nécessité de disposer d'un matériel dédié. Si l'on veut décrire comment créer une distribution Linux pour une autre architecture que le PC/x86 classique, il faudra disposer de la carte adéquate. De même, si l'on veut développer une application pour cette même architecture, on devra également disposer du matériel.

Il y a à cela plusieurs inconvénients :

- Le coût, car les cartes industrielles sont souvent onéreuses. Dans d'autres cas, on devra attendre que la carte soit terminée pour commencer à travailler, et bien entendu, elle ne sera pas longtemps disponible, le matériel étant rare par nature !
- Le fait de devoir transporter du matériel dédié, coûteux, et souvent fragile.
- La difficulté de travailler dans des lieux insolites et/ou mobiles (hôtel, train, etc.)

L'utilisation d'un émulateur comme QEMU résout un grand nombre de problèmes. Il ne coûte rien, les sources sont disponibles, il est facile à installer sur les principaux systèmes d'exploitation (Linux, Windows, Mac OS X, UNIX). Le cauchemar de l'installation d'une salle de travaux pratiques pour un cours « Linux embarqué » s'évanouit alors tels les bénéfices virtuels des banquiers malhonnêtes. QEMU est d'ores et déjà utilisé comme un puissant outil de test et de validation automatique pour des environnements contraints tels que le transport (ferroviaire et aéronautique) ou la défense.

Avant toute chose, il faut installer la distribution binaire de QEMU, même si l'on peut bien entendu le compiler à partir des sources. Dans le cas d'une distribution Fedora, il suffit d'exécuter la commande suivante :

```
$ sudo yum install qemu
```

Pour une distribution Debian, il suffira de faire :

```
$ sudo apt-get install qemu
```

Une fois l'installation effectuée, nous disposons de la commande **qemu**, mais elle correspond à l'architecture x86. Dans notre cas, nous devrons utiliser la commande **qemu-system-arm**.

```
$ qemu-system-arm
QEMU PC emulator version 0.10.6 (qemu-kvm-0.10.6), Copyright (c) 2003-
2008 Fabrice Bellard
usage: qemu [options] [disk_image]

'disk_image' is a raw hard image image for IDE hard disk 0

Standard options:
-h or -help     display this help and exit
-M machine      select emulated machine (-M ? for list)

...
```

La première option (**-M**) de la commande nous permet d'en savoir plus sur les architectures émulées.

```
$ qemu-system-arm -M ?
Supported machines are:
integratorcp ARM Integrator/CP (ARM926EJ-S) (default)
versatilepb ARM Versatile/PB (ARM926EJ-S)
versatileab ARM Versatile/AB (ARM926EJ-S)
realview   ARM RealView Emulation Baseboard (ARM926EJ-S)
akita      Akita PDA (PXA270)
spitz      Spitz PDA (PXA270)
borzoi     Borzoi PDA (PXA270)
terrier    Terrier PDA (PXA270)
sx1-v1     Siemens SX1 (OMAP310) V1
sx1        Siemens SX1 (OMAP310) V2
cheetah    Palm Tungsten|E aka. Cheetah PDA (OMAP310)
n800       Nokia N800 tablet aka. RX-34 (OMAP2420)
n810       Nokia N810 tablet aka. RX-44 (OMAP2420)
lm3s811evb Stellaris LM3S811EVB
lm3s6965evb Stellaris LM3S6965EVB
connex     Gumstix Connex (PXA255)
verdex     Gumstix Verdex (PXA270)
mainstone  Mainstone II (PXA27x)
musicpal   Marvell 88w8618 / MusicPal (ARM926EJ-S)
tosa       Tosa PDA (PXA255)
```

Les noms ci-dessus correspondent à des cartes du marché. QEMU permet donc de simuler entièrement (ou presque) de nombreuses cartes industrielles. Dans notre cas, nous avons choisi la carte ARM Versatile-PB car elle est très bien supportée par Linux et par QEMU. En effet, la majorité des périphériques de la carte (port série, contrôleur Ethernet, contrôleur graphique frame-buffer) sont émulés par QEMU.

2.2 Test de la distribution Linux produite

Le test de l'image sous QEMU s'effectue par la commande suivante :

```
$ qemu-system-arm -M versatilepb -m 16 -kernel  arch/arm/boot/zImage
-initrd /tmp/rootfs_arm.gz
```

L'option **-m 16** indique que nous allouons 16 Mo à l'architecture émulée. L'émulateur affiche alors l'écran ci-dessous. Le temps de démarrage est très rapide, environ 2 secondes !

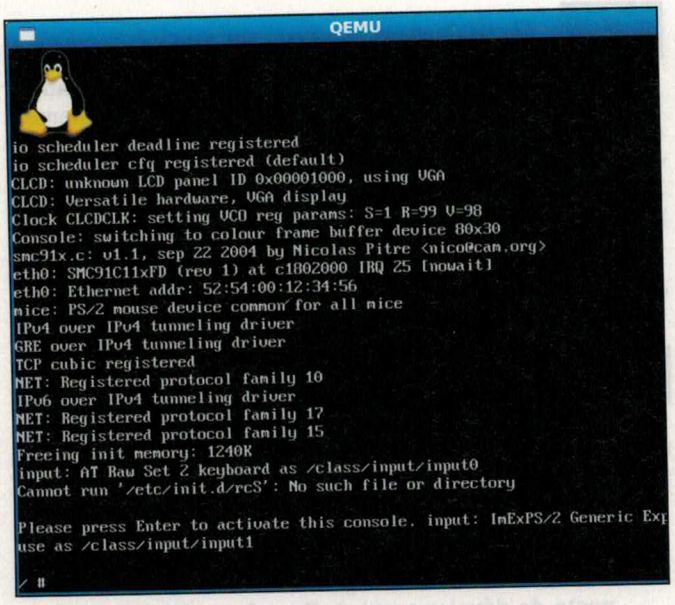


Figure 1 : Exécution de l'image Linux-ARM9 dans QEMU

Nous remarquons que le fichier de démarrage **/etc/init.d/rcS** de Busybox est inexistant. De même, le système de fichiers **/proc** n'est pas monté, ce qui limite les possibilités, mais le système est fonctionnel. Nous remarquons également la détection par le noyau Linux de contrôleur Ethernet SMC91C11xFD de la carte émulée. L'utilisation de la commande **free** montre que le système utilise bien 16 Mo de RAM.

2.3 Amélioration de l'image

Il serait simple d'améliorer le système en ajoutant les éléments suivants :

- montage des systèmes de fichiers **/proc** et **/sys** ;
- un fichier **/etc/fstab** ;

```
proc   /proc  proc  defaults      00
sysfs  /sys   sysfs  defaults     00
```

- un script de démarrage **/etc/init.d/rc.S**.

```
#!/bin/sh
mount -t proc /proc

echo
echo "Busybox + QEMU ARM9 demo"
echo
uptime
```

Bien entendu, il faut de nouveau créer l'image **INITRD** avant d'effectuer un nouveau test.

Ces quelques manipulations démontrent que la mise en place d'une distribution réellement fonctionnelle et pouvant être maintenue dans le temps est une tâche ardue faisant apparaître quelques problèmes :

- la mise en place d'une procédure de génération et d'installation automatisée ;
- les problèmes d'adaptation (portage) des composants ;
- la prise en compte de l'évolution des composants ;
- les dépendances entre les composants ;
- la prise en main de la distribution par des non-spécialistes Linux.

Nous en déduisons, comme le disait souvent le grand Fox Mulder, que la vérité est ailleurs :-)

Par contre, le test réalisé est d'un grand intérêt pédagogique, car il permet de comprendre parfaitement les rouages de la construction et du fonctionnement d'une distribution Linux, même si l'approche n'est pas compatible avec des contraintes de production et de maintenance.

Remarque

Lors des sessions de formation « Linux embarqué » dispensées par l'auteur, les auditeurs subissent la construction manuelle de la distribution, c'est à mon avis un excellent exercice !

Dans la suite de l'article, nous allons réaliser le même type de distribution en utilisant Buildroot et nous pourrons apprécier la facilité avec laquelle nous arriverons à un résultat beaucoup plus avancé.

3 BUILDROOT, UNE INTRODUCTION

Le projet Buildroot est depuis toujours lié à uClibc (µ-C-libc, http://www.uclibc.org). Le but d'uClibc est de fournir une solution alternative à la bibliothèque libc de référence Glibc (GNU-libc). La Glibc est très complète, mais elle est également très complexe et volumineuse, ce qui n'est pas forcément en adéquation avec les contraintes de l'informatique embarquée.

La bibliothèque uClibc est plus de quatre fois plus légère que la Glibc et ses fonctionnalités sont largement suffisantes pour la plupart des projets Linux embarqué industriels. De plus, elle est nativement développée pour les architectures embarquées, ce qui n'est pas le cas de la Glibc.

Pour effectuer des tests uClibc, les développeurs du projet ont depuis longtemps mis en place un outil de construction de distribution, c'est ainsi qu'est né le projet Buildroot sur http://buildroot.uclibc.org. Du fait de son statut d'outil de test, le projet n'était pas réellement diffusé, sauf sous forme de dépôt SVN. De plus, la maintenance était très chaotique avec des corrections disparates suivant les architectures.

L'utilisation de Buildroot pour un projet industriel était possible, mais l'absence de version officielle rendait l'approche difficile puisque la distribution était basée sur un « snapshot » (une version donnée) du dépôt SVN. Cependant, de nombreuses sociétés utilisent Buildroot depuis de nombreuses années, citons THALES, ATMEL, GUMSTIX, ARMADEUS, et certainement bien d'autres anonymes.

3.1 Qu'est-ce que Buildroot ?

Buildroot n'est pas une distribution Linux, c'est un ensemble de fichiers `Makefile` et de scripts shell permettant de construire les outils de production puis la distribution cible à partir des données d'entrée, soit :

- Les archives des sources des composants obtenues directement dans leur version non adaptée sur les sites des projets. Les archives sont chargées lors de la première utilisation de Buildroot, par défaut sur le sous-répertoire `dl`.
- La configuration fournie par l'utilisateur : l'architecture cible, les composants à intégrer, la version du compilateur et d'uClibc, etc.
- Les différents patchs fournis par Buildroot. Outre le système de génération configurable, c'est une valeur ajoutée importante de Buildroot.

Remarque

- En plus de la distribution cible, Buildroot est capable de générer une chaîne de compilation croisée basée sur uClibc. Si la chaîne doit être basée sur Glibc, on pourra l'utiliser en tant que chaîne externe (*external toolchain*), mais elle devra être produite par un autre outil. Cette solution sera évoquée dans le deuxième article « Buildroot avancé ».
- Buildroot peut également prendre en charge la compilation du noyau Linux.
- Buildroot ne produit pas de paquetage individuel (exemple : au format IPKG) pour chaque composant, c'est une de ses faiblesses par rapport à d'autres systèmes, comme OpenEmbedded ou Openwrt.

3.2 Buildroot aujourd'hui

Depuis début 2009, Buildoot est officiellement maintenu par Peter Korsgaard. En France, Thomas Petazzoni est également un contributeur très actif. Des versions officielles sont publiées environ tous les trois mois, soit pour 2009 : 2009.2, 2009.5, 2009.8, 2009.11. Le dépôt principal du projet utilise désormais le gestionnaire de version Git (http://git.buildroot.net/buildroot).

Bien entendu, Buildroot est toujours un logiciel libre diffusé sous licence GPL (v2).

3.3 Buildoot et ses concurrents

Buildroot n'est pas le seul outil de création de distribution embarquée. Nous avons cité en introduction plusieurs outils similaires. En résumé, nous pouvons dire que Buildroot se veut :

- Léger, ce qui n'est pas le cas d'OpenEmbedded, qui est puissant et évolutif (plus que Buildroot), mais difficile d'accès et très gourmand en ressources sur le poste de développement. De même, OpenEmbedded ne fournit pas d'outil de configuration de type `make menuconfig` et la configuration s'effectue uniquement par l'édition de fichiers.
- Généraliste, alors qu'Openwrt est plutôt dédié aux IAD – *Internet Access Devices*. Cependant, Openwrt est certainement le projet le plus proche de Buildroot et issu de ce dernier. Openwrt est cependant capable de gérer des paquetages binaires (IPKG/OPKG) que l'on peut installer sur la cible a posteriori, ce que ne sait pas encore faire Buildroot.
- Non lié à un constructeur ou une entreprise, à la différence de la majorité des outils fournis avec les cartes, même si ceux-ci sont parfois dérivés de Buildroot ou d'autres outils équivalents.

4 INSTALLATION ET UTILISATION

Comme nous l'avons précisé précédemment, Buildroot n'est pas une distribution Linux et l'archive du projet n'est dont pas très volumineuse (environ 5 Mo). Pour l'installer, on peut se rendre sur http://buildroot.uclibc.org/download.html et charger la dernière version à ce jour (2009.11).

L'installation sur un répertoire de test est triviale :

```
$ tar xjvf ~/Téléchargement/buildroot-2009.11.tar.bz2
$ cd buildroot-2009.11
```

REMARQUE : Il n'est pas recommandé de déplacer le répertoire une fois que Buildroot est utilisé.

La configuration est basée sur le même outil que le noyau Linux et Busybox. On peut utiliser un outil de configuration en mode texte (`menuconfig`) ou basé sur Qt3 (`xconfig`). Personnellement, j'ai toujours utilisé la commande en mode texte et je ne suis pas certain du bon fonctionnement du configurateur Qt.

```
$ make menuconfig
```

On obtient alors l'écran suivant :

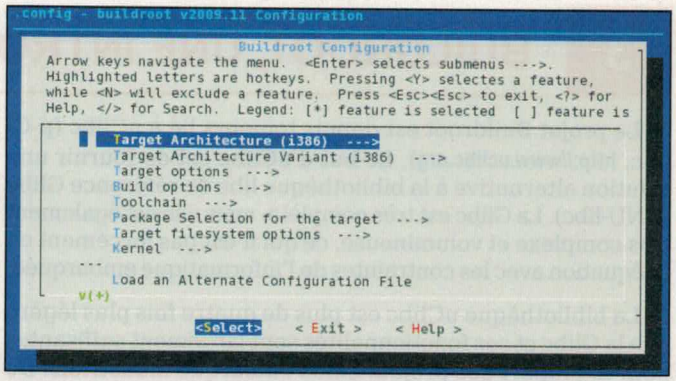


Figure 2 : Ecran de configuration Buildroot

La liste des rubriques au premier niveau est assez réduite :

- *Target Architecture* : le type de CPU, exemple : arm ;

- *Target Architecture Variant* : le sous-type, exemple : ARM920T ;
- *Target options* : les options spécifiques au constructeur (exemple : Atmel) ou à la carte (nom du système, bannière d'accueil, etc.) ;
- *Build options* : les options des outils de production utilisés par Buildroot (exemples : wget ou svn, localisation des sites de téléchargement des archives sources) ;
- *Toolchain* : le paramétrage de la chaîne de compilation croisée ;
- *Package selection for the target* : sélection des paquetages à installer sur la distribution cible, avec gestion des dépendances ;
- *Target filesystem options* : type d'image cible à créer en vue de l'installation sur la cible (exemples : tar, ext2, jffs2, cpio, etc.) ;
- *Kernel* : les paramètres du noyau Linux à compiler, même si par défaut Buildroot ne compile pas le noyau.

4.1 Un premier test de génération

Le but de ce premier test est d'utiliser au maximum les configurations par défaut afin d'arriver à un résultat proche du test manuel décrit au début de l'article. En dépit de cela, nous verrons que la distribution produite est plus complète pour un effort bien moindre. Pour cela, nous devons tout d'abord configurer l'environnement Buildroot.

En sélectionnant la première rubrique (Target Architecture), on note que la majorité des architectures utilisées dans l'industrie sont disponibles : arm, x86, mips, PowerPC, SuperH, etc.

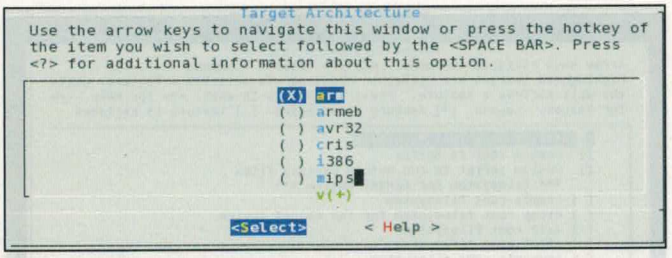


Figure 3 : Sélection d'architecture cible

Si l'on sélectionne arm, on peut alors dérouler les différentes variantes de l'architecture dans le deuxième menu (Target Architecture Variant), comme l'ARM920T, ARM920T, etc.

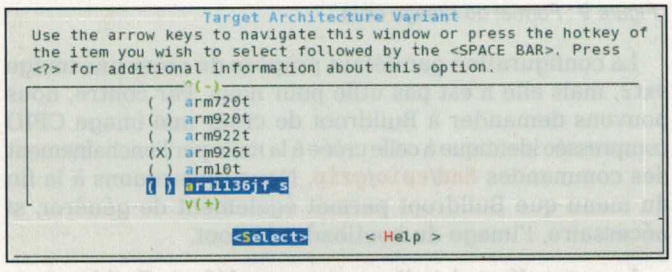


Figure 4 : Variantes du processeur ARM

Dans le cas présent, le processeur de la carte émulée est un ARM926EJ-S.

La sélection de l'architecture ARM fait apparaître un nouveau menu *Target ABI* de choix de l'ABI (*Application Binary Interface*). Nous avons le choix entre EABI (*Embedded ABI*) et OABI (*Old ABI*). L'émulateur QEMU supportant uniquement OABI, il est nécessaire de sélectionner cette option.

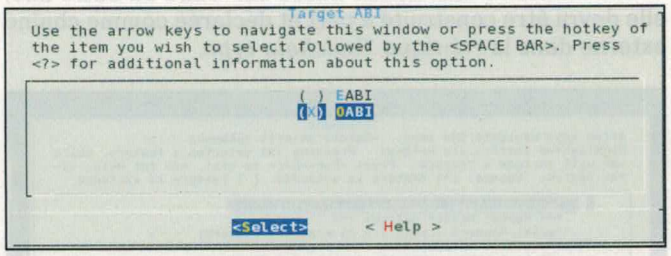


Figure 5 : Sélection de l'OABI

Le menu Target options nous permet uniquement de saisir le nom du système (soit versatilepb) et la bannière d'accueil.

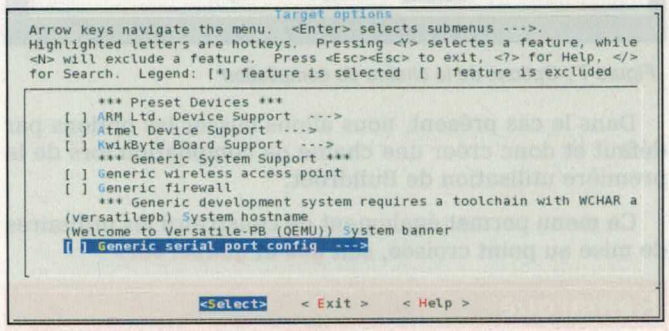


Figure 6 : Options de la cible

Remarque

La dernière option du menu (*Generic serial port config*) est importante, car elle permet de spécifier quel périphérique sera utilisé pour la console système. Cette option n'a pas d'importance pour l'instant, car la carte Versatile-PB émulée dispose d'un *framebuffer*. De ce fait, la console utilisera une console virtuelle (**/dev/tty1**) exactement comme sur un PC/x86. Dans le cas d'une véritable carte ne disposant pas de framebuffer, la console utilisera un port série physique (UART) associé à une entrée de type **/dev/ttyS0** ou autre fichier spécial dépendant de la cible.

Si l'on ne renseigne pas cette option, on n'obtiendra jamais l'affichage de la bannière de **login**. Nous verrons ce point plus en détail lors du test sur une carte réelle, dans l'article « Buildroot avancé ».

Le menu Build options n'a pas à être modifié. Cependant, nous pouvons constater en entrant dans ce menu qu'il dispose d'une rubrique *Mirror and Download locations*. Dans cette rubrique, nous constatons que le projet Buildroot héberge des copies des archives des différents composants sur **http://buildroot.net/downloads/sources**. Cette fonctionnalité est très utile en cas d'indisponibilité du site officiel hébergeant le composant ou de disparition de la version de composant utilisée par Buildroot.

Le menu Toolchain permet de générer par défaut une chaîne de compilation croisée. Buildroot étant dérivé du projet uClibc, la chaîne produite sera OBLIGATOIREMENT basée sur uClibc. Comme toute chaîne de compilation, elle est également dépendante du noyau Linux.

Si l'on désire une chaîne basée sur Glibc ou autre libc, elle devra être construite à part et déclarée comme chaîne externe dans la rubrique *Toolchain type*.

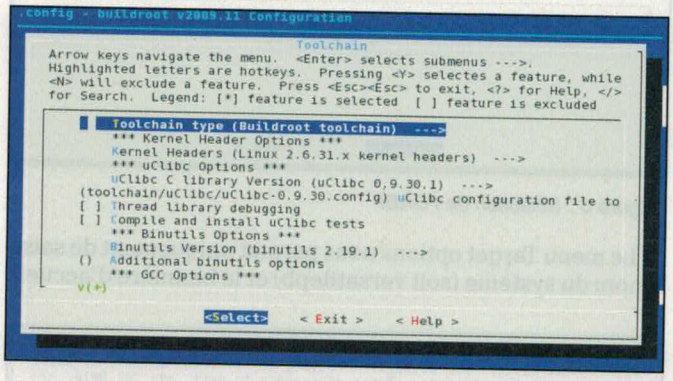


Figure 7 : Options de la chaîne de compilation

Dans le cas présent, nous allons laisser les options par défaut et donc créer une chaîne de compilation lors de la première utilisation de Buildroot.

Ce menu permet également de construire les utilitaires de mise au point croisée, soit **gdb** et **gdbserver**.

Remarque

- La version des en-têtes (*Kernel Headers*) n'est pas forcément la même que celle du noyau utilisé par la cible. Cependant, si le noyau de la cible est beaucoup plus ancien que le noyau et la libc utilisés par la chaîne de compilation, il est fortement probable que cela conduise à des erreurs au démarrage du système (message « *kernel too old* » et arrêt du système). D'un autre côté, l'intégration d'un noyau ancien dans une chaîne récente n'est pas forcément vouée au succès, il faut donc parfois jouer finement et faire plusieurs essais de combinaison de versions.
- L'utilisation d'une chaîne externe peut être une solution au problème précédent. Cependant, les dernières versions de Buildroot imposent que la chaîne dispose de l'option **--sysroot**. Cette option permet de spécifier le chemin d'accès des bibliothèques et des en-têtes, elle est disponible uniquement sur le version 4 de GCC.

Le menu Package Selection for the target constitue une forte valeur ajoutée de Buildroot : la sélection des différents composants gère automatiquement les dépendances, tant au niveau cible que poste de développement. Par exemple, la sélection d'un outil d'affichage d'image comme **fbv** impliquera automatiquement la compilation de la bibliothèque LibJPEG. De même, le choix d'un type de format pour le root-filesystem cible impliquera l'installation sur la machine de développement des outils permettant de produire l'image (exemple : le format jffs2 et l'outil **mkfs.jffs2**).

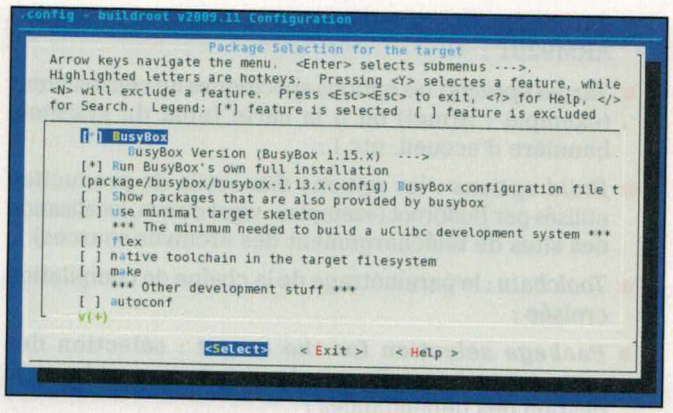


Figure 8 : Sélection des paquetages

Dans le cas présent, seul le paquetage Busybox est sélectionné et l'image produite sera donc assez proche de ce que nous avions réalisé à la main, avec cependant quelques fonctionnalités indispensables pour un véritable système Linux. Nous remarquons que la configuration de Busybox peut être modifiée grâce à un fichier précisé en paramètre (par défaut, c'est le fichier **package/busybox/busybox-1.13.x.config**).

Il est également possible d'accéder à la configuration de Busybox en utilisant la commande :

```
$ make busybox-menuconfig
```

Nous reviendrons ultérieurement sur les possibilités de configuration de ce menu.

Le menu Target filesystem options permet de préciser sous quelle forme l'image du root-filesystem sera produite.

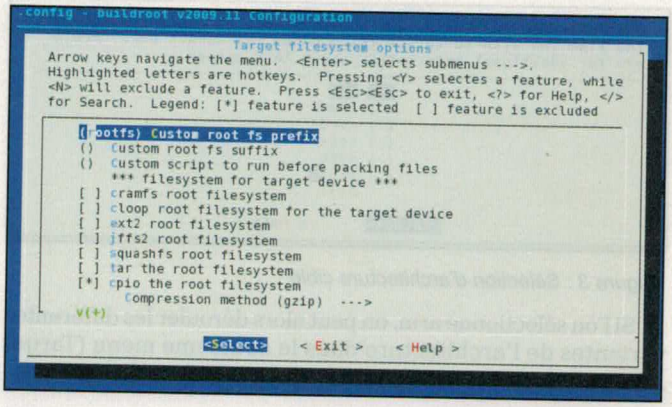


Figure 9 : Format de l'image cible

La configuration par défaut propose de créer une image ext2, mais elle n'est pas utile pour nous. Par contre, nous pouvons demander à Buildroot de créer une image CPIO compressée identique à celle créée à la main par l'enchaînement des commandes **find**/**cpio**/**gzip**. Nous remarquons à la fin du menu que Buildroot permet également de générer, si nécessaire, l'image du bootloader U-Boot.

Le menu Kernel indique que par défaut, Buildroot ne compile pas le noyau Linux pour la cible. Nous utiliserons le noyau compilé au début de l'article puisque l'architecture cible est identique.

On peut enfin quitter Buildroot, sans oublier de sauver le précieux fichier de configuration **.config** que nous avons créé.

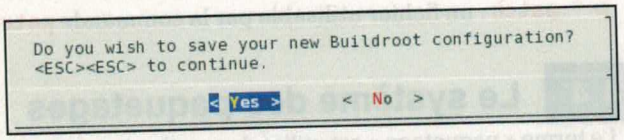


Figure 10 : Sauvegarde de la configuration

Le format du fichier **.config** est identique à celui du noyau Linux. Les noms des variables commencent par **BR2**.

```
#
# Automatically generated make config: don't edit
# Mon Jan 18 14:03:08 2010
#
BR2_HAVE_DOT_CONFIG=y
BR2_VERSION="2009.11"
# BR2_alpha is not set
BR2_arm=y
```

On peut alors lancer la compilation en utilisant la commande **make**. La première compilation est longue (minimum 20 à 30 minutes sur un PC puissant), car il est nécessaire d'effectuer des tâches préliminaires :

- chargement des archives vers le répertoire **dl** ;
- génération de la chaîne croisée.

Au niveau espace disque, une première compilation occupe environ 2 Go sur le disque.

Remarque

Nous conseillons d'utiliser la variable **V** (*Verbose*) déjà disponible pour la compilation du noyau Linux et de Busybox. Cela permet de suivre précisément les étapes de la compilation.

```
$ make V=1
```

A l'issue de la compilation, on obtient le résultat dans le répertoire **output**.

```
$ ls -l output/images
total 1684
-rw-r--r-- 1 pierre users 1164800 janv. 17 18:14 rootfs.arm.cpio
-rw-r--r-- 1 pierre users  554131 janv. 17 18:14 rootfs.arm.cpio.gz
```

On peut alors tester la nouvelle distribution avec QEMU.

```
$ qemu-system-arm -M versatilepb -m 16 -kernel <kernel-path>arch/arm/
boot/zImage -initrd output/images/rootfs.arm.cpio.gz
```

On obtient l'image ci-après. Contrairement à notre distribution minimaliste, l'image générée par Buildroot gère l'authentification des utilisateurs. On peut se logger en tant que root et il n'y a pas de mot de passe par défaut (voir Figure 11).

4.2 Création d'une configuration prédéfinie

L'archive Buildroot 2009.11 contient plusieurs fichiers de configuration prédéfinie pour les cartes officiellement supportées par le projet. L'avantage évident est d'éviter a

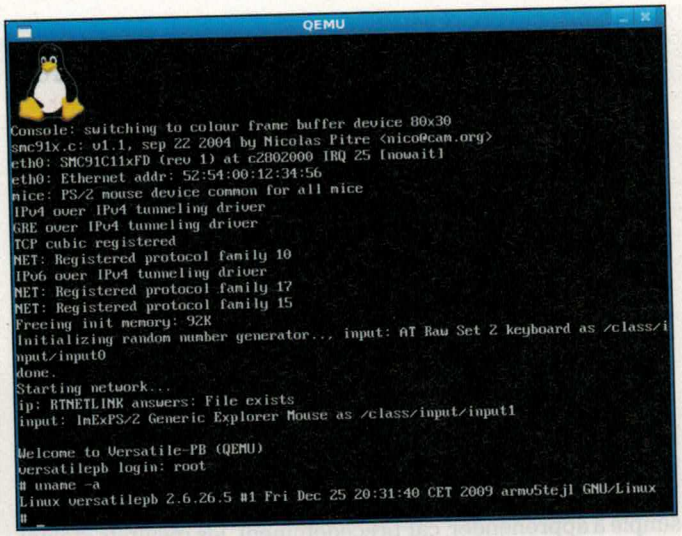


Figure 11 : Première distribution Buildroot dans QEMU

posteriori d'effectuer le travail que nous avons réalisé dans les paragraphes précédents. Pour cela, il suffit simplement de copier le fichier **.config** dans le répertoire **configs**.

```
$ cp .config configs/QEMU_versatilepb_uclibc_defconfig
```

Pour un nouvel utilisateur recevant la version 2009.11 ainsi modifiée, la production de l'image se résumera aux commandes suivantes :

```
$ make QEMU_versatilepb_uclibc_defconfig
$ make
```

4.3 Un aperçu de la structure de Buildroot

Dans le paragraphe précédent, nous avons décrit comment arriver rapidement à un résultat utilisable. Cependant, Buildroot est un outil puissant et configurable, il est donc nécessaire d'étudier d'un peu plus près sa structure afin de l'utiliser au mieux. Pour faciliter la compréhension, les détails de la structure de Buildroot seront fournis au fur et à mesure du déroulement de l'article.

Contrairement à d'autres outils (comme OpenEmbedded, qui est basé sur **bitbake**, un script Python), le moteur de Buildroot est basé sur des commandes standards de Linux : **gmake**, **bash**, **find**, etc.

Au premier niveau des sources, on trouve les fichiers suivants.

```
$ls -l
total 100
-rw-r--r--   1 pierre users 11458 déc.   115:27 CHANGES
-rw-r--r--   1 pierre users 10130 déc.   115:27 Config.in
drwxr-xr-x   2 pierre users  4096 déc.   115:27 configs
-rw-r--r--   1 pierre users 17987 déc.   115:27 COPYING
drwxr-xr-x   3 pierre users  4096 janv. 18 10:01 docs
-rw-r--r--   1 pierre users 19497 déc.   115:27 Makefile
drwxr-xr-x 306 pierre users 12288 déc.   115:27 package
drwxr-xr-x   3 pierre users  4096 déc.   115:27 scripts
drwxr-xr-x  21 pierre users  4096 déc.   115:27 target
-rw-r--r--   1 pierre users   806 déc.   115:27 TODO
drwxr-xr-x  13 pierre users  4096 déc.   115:27 toolchain
```

Les principaux répertoires sont :

- **package** : contient les paquetages gérés par Buildroot. A terme, on peut ajouter des sous-répertoires à **package** si l'on désire ajouter des composants (voir à la fin de l'article).
- **target** : définit les paramètres dépendant de la cible (type d'image à générer, configuration matérielle). Dans la suite de l'article, nous donnerons un exemple d'ajout de cible matérielle réelle (répertoire **target/device**).
- **toolchain** : définit les paramètres de construction de la chaîne croisée. Le répertoire contient également les paramètres pour l'utilisation d'une chaîne externe.

Lorsque la compilation est terminée, on a de plus le répertoire **dl** et surtout le répertoire **output**, qui contient le résultat de la compilation. Ce répertoire est une nouveauté de la version 2009.11, la structure est de ce fait beaucoup plus simple à appréhender, car précédemment, les résultats étaient produits sur plusieurs répertoires (**project_build_<arch>**, **build_<arch>**, **toolchain**, **toolchain_build_<arch>**) si **<arch>** était le nom de l'architecture cible (exemple : arm). Dans le cas présent, on a :

```
$ ls -l output
total 28
drwxr-xr-x 17 pierre users 4096 janv. 20 18:07 build
drwxr-xr-x  3pierre users 4096 janv. 20 18:05 host
drwxr-xr-x  2pierre users 4096 janv. 20 17:52 images
drwxr-xr-x  5pierre users 4096 janv. 20 17:52 staging
drwxr-xr-x  2pierre users 4096 janv. 20 18:05 stamps
drwxr-xr-x 16 pierre users 4096 janv. 20 18:04 target
drwxr-xr-x 19 pierre users 4096 janv. 20 18:02 toolchain
```

- **build** : répertoire de construction des différents composants (paquetages, noyau Linux, etc.), hormis la chaîne de compilation croisée ;
- **host** : contient les outils produits par Buildroot, mais utilisés sur le poste de développement (*host*) ;
- **images** : contient les images produites à installer sur la cible ;
- **staging** : contient la chaîne de compilation croisée si elle est construite par Buildroot ;
- **target** : contenu du root-filesystem de la cible, sauf le répertoire **dev**, traité ultérieurement pour produire l'image du root-filesystem dans le répertoire images ;
- **toolchain** : répertoire de construction de la chaîne de compilation croisée.

Les principaux fichiers utilisés sont les suivants :

- **Config.in** : un fichier décrivant l'interface de configuration (géré par **make menuconfig**). La modification de l'outil de configuration (exemple : pour ajouter une nouvelle carte) passe par l'ajout de fichiers de ce type ;
- **Makefile.in.*** : un fichier inclus dans un autre Makefile (ou **Makefile.in.***) par une directive **include** de **gmake**. La modification de Buildroot nécessite d'ajouter des fichiers **Makefile.in**, associés à des **Config.in** ;
- **Makefile** : fichier **Makefile** principal de Buildroot ;
- ***.mk** : le plus souvent, un fichier **Makefile** décrivant la compilation d'un composant dans le répertoire **package** ;
- ***.sh** : un shell-script. Les scripts sont le plus souvent appelés dans les fichiers **Makefile.*** ;
- ***.patch** : un fichier utilisable par la commande **patch**.

4.4 Le système des paquetages

Le terme « paquetage » est utilisé de manière impropre, car Buildroot ne gère PAS réellement de paquetage (exemples : fichiers **.rpm**, **.deb**, **.ipk**). En l'occurrence, il est IMPOSSIBLE d'ajouter proprement un nouveau composant à une image produite par Buildroot sans produire une nouvelle image par **make**.

Cependant, l'un des gros avantages de Buildroot est de fournir un grand nombre de composants adaptés aux architectures embarquées. Le répertoire **package** contient un peu plus de 300 entrées. L'adaptation d'un composant pour la compilation croisée n'est pas toujours simple, malgré l'utilisation de plus en plus fréquente d'outils comme Autotools.

En effet, il faut également que l'auteur du projet ait prévu la possibilité d'utiliser un environnement croisé. Si le composant est complexe, ce n'est pas forcément trivial. A titre d'exemple, si l'on tente de compiler la bibliothèque Kerberos-1.6.3 pour cible arm, on obtient une erreur.

```
$ ./configure --host=arm-linuxconfigure: WARNING: If you wanted to set
the --build type, don't use --host.
    If a cross compiler is detected then cross compile mode will be used.
configure: creating cache ./config.cache
checking for arm-linux-gcc... arm-linux-gcc
...
checking whether pragma weak references are supported... yes
checking for constructor/destructor attribute support... configure:
error: Cannot test for constructor/destructor support when cross compiling
```

Il est nécessaire de passer quelques options un peu « ésotériques » pour arriver à un bon résultat !

```
$ cross_compiling=${cross_compiling=yes,yes} \
    krb5_cv_attr_constructor_destructor=${krb5_cv_attr_constructor_
destructor=yes,yes} \
    ac_cv_func_regcomp=${ac_cv_func_regcomp=yes,yes} \
    ac_cv_printf_positional=${ac_cv_printf_positional=yes,yes} \
    ac_cv_file__etc_environment=${ac_cv_file__etc_environment=no,no} \
    ac_cv_file__etc_TIMEZONE=${ac_cv_file__etc_TIMEZONE=no,no} \
    \
    enable_thread_support=${enable_thread_support=no,no} \
    enable_thread=${enable_thread_support=no,no} \
      KRB5_AC_ENABLE_THREADS=${KRB5_AC_ENABLE_THREADS=no,no} ./configure
--disable-thread-support --enable-shared --without-krb4 --without-tcl \
    --disable-ipv6 --host=arm-linux
configure: WARNING: If you wanted to set the --build type, don't use --host.
    If a cross compiler is detected then cross compile mode will be used.
configure: loading cache ./config.cache
checking for arm-linux-gcc... arm-linux-gcc
...
configure: creating ./config.status
config.status: creating ./Makefile
config.status: creating resolve/Makefile
config.status: creating asn.1/Makefile
config.status: creating create/Makefile
config.status: creating hammer/Makefile
config.status: creating verify/Makefile
config.status: creating gssapi/Makefile
config.status: creating dejagnu/Makefile
config.status: creating threads/Makefile
config.status: creating shlib/Makefile
config.status: creating gss-threads/Makefile
config.status: creating misc/Makefile
```

Autre problème : de nombreux composants dépendent d'autres composants – comme des bibliothèques – ce qui rend l'adaptation manuelle très fastidieuse. Là aussi, Buildroot traite le problème de manière transparente pour l'utilisateur final.

Un répertoire de paquetages contient au minimum deux fichiers :

- le fichier **Config.in** décrit l'entrée dans l'outil de configuration (**menuconfig**) ;
- le fichier **.mk** décrit le Makefile de construction du composant.

Au niveau précédent (répertoire **package**), un fichier **Config.in** utilise les différentes entrées.

```
comment "Other stuff"
source "package/at/Config.in"
source "package/beecrypt/Config.in"
source "package/berkeleydb/Config.in"
...
```

Nous rappelons que les paquetages sont compilés dans le répertoire **output/build**.

4.4.1 Exemple du paquetage fbv

Dans la suite, nous pouvons prendre l'exemple de l'utilitaire **fbv** permettant d'afficher des images sur le framebuffer de Linux. Le fichier **Config.in** contient les lignes suivantes :

```
config BR2_PACKAGE_FBV
  bool "fbv"
  select BR2_PACKAGE_LIBPNG
  select BR2_PACKAGE_JPEG
  select BR2_PACKAGE_LIBUNGIF
  help
    fbv is a very simple graphic file viewer for the framebuffer console,
    capable of displaying GIF, JPEG, PNG and BMP files using libungif,
    libjpeg and libpng.
    http://freshmeat.net/projects/fbv
```

Nous remarquons la commande **select**, qui force la validation des options nécessaires à la compilation du composant (**BR2_PACKAGE_LIBPNG**, etc.). Ce fichier est inclus dans **package/Makefile.in** par la ligne :

```
source "package/fbv/Config.in"
```

Le fichier **fbv.mk** est plus compliqué. La première partie du fichier décrit les différentes constantes utilisées. La constante **FBV_TARGET_BINARY** indique le chemin d'accès à la commande sur le root-filesystem cible.

```
#############################################################
#
# fbv
#
#############################################################
FBV_VERSION:=1.0b
FBV_SOURCE:=fbv-$(FBV_VERSION).tar.gz
FBV_SITE:=http://s-tech.elsat.net.pl/fbv
FBV_DIR:=$(BUILD_DIR)/fbv-$(FBV_VERSION)
FBV_CAT:=$(ZCAT)
FBV_BINARY:=fbv
FBV_TARGET_BINARY:=usr/bin/$(FBV_BINARY)
```

La deuxième partie décrit les règles de production. La commande **call** du Makefile permet d'exécuter une macro. Ce principe est très fréquemment utilisé dans Buildroot. Dans le cas présent, la macro **DOWNLOAD** est définie dans **package/Makefile.autotools.in**.

```
$(DL_DIR)/$(FBV_SOURCE):
  $(call DOWNLOAD,$(FBV_SITE),$(FBV_SOURCE))

fbv-source: $(DL_DIR)/$(FBV_SOURCE)
```

Buildroot utilise également des fichiers spéciaux (**.unpacked**, **.patched**, **.configured**, **.applied_patches_list**, ...). Le but (au sens de l'outil **make**) associé décrit l'ordre de traitement. Cependant, les noms utilisés dans les différents fichiers **.mk** peuvent varier suivant l'auteur, ce qui peut parfois poser un problème de compréhension.

Dans le cas étudié, on extrait les sources et on applique des patchs avant de configurer (but **.unpacked**). Les fichiers de patch sont situés dans le répertoire du composant et doivent correspondre à l'expression régulière décrite ci-dessous.

```
$(FBV_DIR)/.unpacked: $(DL_DIR)/$(FBV_SOURCE)
  $(FBV_CAT) $(DL_DIR)/$(FBV_SOURCE) | tar -C $(BUILD_DIR) $(TAR_OPTIONS) -
  toolchain/patch-kernel.sh $(FBV_DIR) package/fbv/ \
    fbv-$(FBV_VERSION)\*.patch fbv-$(FBV_VERSION)\*.patch.$(ARCH)
  touch $@
```

Lorsque cette étape est réalisée, on utilise le script **configure**, le but **.configured** dépend du but précédent **.unpacked**. Bien évidemment, cette étape n'existe pas si le composant n'utilise pas Autotools.

```
$(FBV_DIR)/.configured: $(FBV_DIR)/.unpacked
  (cd $(FBV_DIR); rm -f config.cache; \
    $(TARGET_CONFIGURE_OPTS) \
    $(TARGET_CONFIGURE_ARGS) \
    ./configure \
    --prefix=/usr \
    --libs="-lz -lm" \
  )
  touch $@
```

Finalement, la production du binaire dépend du but **.configured** .

```
$(FBV_DIR)/$(FBV_BINARY): $(FBV_DIR)/.configured
  $(MAKE) $(TARGET_CONFIGURE_OPTS) -C $(FBV_DIR)

$(TARGET_DIR)/$(FBV_TARGET_BINARY): $(FBV_DIR)/$(FBV_BINARY)
   install -D $(FBV_DIR)/$(FBV_BINARY) $(TARGET_DIR)/$(FBV_TARGET_BINARY)
```

Cette ligne définit les dépendances de **fbv** par rapport à d'autres composants comme **libpng**, **libjpeg** et **libungif**.

```
fbv: libpng jpeg libungif $(TARGET_DIR)/$(FBV_TARGET_BINARY)
```

Nous trouvons ensuite les buts classiques de nettoyage du répertoire.

```
fbv-clean:
  rm -f $(TARGET_DIR)/$(FBV_TARGET_BINARY)
  -$(MAKE) -C $(FBV_DIR) clean

fbv-dirclean:
  rm -rf $(FBV_DIR)
```

Pour finir, on ajoute le nom du composant à la variable **TARGETS**.

```
#############################################################
#
# Toplevel Makefile options
#
#############################################################
ifeq ($(BR2_PACKAGE_FBV),y)
TARGETS+=fbv
endif
```

4.4.2 Reconstruction des paquetages

Ce point n'est pas le mieux traité par Buildroot, puisque la notion de paquetage au sens strict n'existe pas. Buildroot ne maintient pas véritablement la liste des composants binaires installés sur la cible. En conséquence, le plus simple, et surtout le plus déterministe, est de reconstruire le paquetage en effaçant le répertoire lui correspondant dans **output/build**.

```
$ rm -rf output/build/fbv-1.0b
$ make
```

Certains fichiers **.mk** permettent d'utiliser des solutions, mais ce n'est pas le cas général.

4.4.3 Ajout de paquetages externes

Dans ce paragraphe, nous allons décrire comment ajouter une liste de paquetages extérieurs à Buildroot. Cela peut être utilisé si vous souhaitez ajouter des paquetages non pris en charge ou si vous désirez gérer vos applications dans Buildroot.

Pour cela, il faut tout d'abord que l'application puisse être compilée correctement à l'extérieur de Buildroot en utilisant la chaîne produite (ou la chaîne externe si Buildroot est configuré pour cela). Dans le cas présent, nous partons du principe que nous utilisons la chaîne interne basée sur uClibc. Elle est installée sur **output/staging**, on peut donc ajouter le chemin d'accès par la commande :

```
$ export PATH=$PATH:`pwd`/output/staging/usr/bin
```

Puis éventuellement, si c'est nécessaire :

```
$ export ARCH=arm
$ export CROSS_COMPILE=arm-linux-
```

La chaîne est désormais accessible.

```
$ arm-linux-gcc -v
Using built-in specs.
Target: arm-linux-uclibc
...
gcc version 4.3.4 (GCC)
```

Nous considérons un exemple simpliste de type « Hello World » constitué d'un seul fichier **hello_world.c** et d'un fichier **Makefile**. On peut valider la compilation en testant par la commande :

```
$ make CC=arm-linux-gcc
arm-linux-gcc  -o hello_world  hello_world.c
$ file hello_world
hello_world: ELF 32-bit LSB executable, ARM, version 1, dynamically
linked (uses shared libs), not stripped
```

Bien entendu, le but est d'automatiser la compilation par Buildroot ainsi que l'installation sur le root-filesystem de la cible. Pour cela, on doit modifier le répertoire package en suivant la procédure ci-après.

1. Modifier le fichier **package/Config.in** afin de faire apparaître les nouveaux paquetages (ou groupes de paquetages). Dans notre cas, on ajoute une rubrique *My Applications*, qui apparaîtra à la fin du menu Package Selection for the target, soit :

```
menu "My applications"
source "package/hello_world/Config.in"
endmenu
```

2. Pour chaque paquetage, ajouter un fichier **Config.in**, un fichier **.mk** et éventuellement des fichiers de patch si la compilation du composant dans l'environnement cible le nécessite.

Dans le cas présent, le fichier **package/hello_world/Config.in** est décrit ci-après. On notera que le nouveau paquetage correspond à une nouvelle variable dans le fichier **.config**.

```
config BR2_PACKAGE_HELLOWORLD
  bool "Hello World"
  help
    Hello World application
```

Le fichier **hello_world.mk** semble plus complexe, mais on remarquera qu'il est très proche de l'exemple étudié précédemment (commande **fbv**). La différence est l'absence de but **.configured** sachant que cet exemple n'utilise pas Autotools, donc pas de script **configure**. Ce fichier est donc très générique et pourra être adapté facilement à grand renfort de copier/coller !

Après modification, la configuration des paquetages de Buildroot apparaît comme ci-dessous :

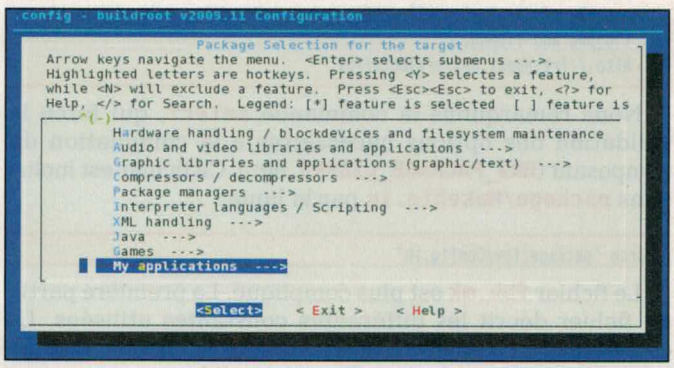


Figure 12 : Menu des paquetages après modification

Après une nouvelle compilation, on peut vérifier si le nouveau composant est bien installé.

```
$ls -l output/target/usr/bin/hello_world
-rwxr-xr-x 1 pierre users 2780 janv. 23 22:50 output/target/usr/bin/
hello_world
$ ls -l output/build/hello_world-1.0/
total 16
-rwxr-xr-x 1 pierre users 5139 janv. 23 22:50 hello_world
-rw-r--r-- 1 pierre users  101 janv. 23 19:22 hello_world.c
-rw-r--r-- 1 pierre users  144 janv. 23 19:24 Makefile
```

Remarque

Le nom de l'archive installée sur le site distant doit correspondre à la variable **HELLOWORLD_SOURCE**.

```
##############################################################
#
# hello_world  command
#
##############################################################
HELLOWORLD_VERSION:=1.0
HELLOWORLD_SOURCE:=hello_world-$(HELLOWORLD_VERSION).tar.gz
HELLOWORLD_SITE:=http://pficheux.free.fr/articles/lmf/Buildroot
HELLOWORLD_DIR:=$(BUILD_DIR)/hello_world-$(HELLOWORLD_VERSION)
HELLOWORLD_CAT:=$(ZCAT)
HELLOWORLD_BINARY:=hello_world
HELLOWORLD_TARGET_BINARY:=usr/bin/$(HELLOWORLD_BINARY)

# Get archive from site
$(DL_DIR)/$(HELLOWORLD_SOURCE):
        $(WGET) -P $(DL_DIR) $(HELLOWORLD_SITE)/$(HELLOWORLD_SOURCE)

# Extract to output/build
$(HELLOWORLD_DIR)/.unpacked: $(DL_DIR)/$(HELLOWORLD_SOURCE)
        $(HELLOWORLD_CAT) $(DL_DIR)/$(HELLOWORLD_SOURCE) | tar -C
        $(BUILD_DIR)\ $(TAR_OPTIONS) -
        touch $@

# Build program
$(HELLOWORLD_DIR)/$(HELLOWORLD_BINARY): $(HELLOWORLD_DIR)/.unpacked
        $(MAKE) $(TARGET_CONFIGURE_OPTS) -C $(HELLOWORLD_DIR)
        touch -c $@
```

```
#Install it to target root-fs
$(TARGET_DIR)/$(HELLOWORLD_TARGET_BINARY): $(HELLOWORLD_
DIR)/$(HELLOWORLD_BINARY)
  install -D $(HELLOWORLD_DIR)/$(HELLOWORLD_BINARY) $(TARGET_
DIR)/$(HELLOWORLD_TARGET_BINARY)

# Dependencies: only uclibc + fakeroot
hello_world:  uclibc  host-fakeroot  $(TARGET_DIR)/$(HELLOWORLD_TARGET_
BINARY)

hello_world-source: $(DL_DIR)/$(HELLOWORLD_SOURCE)

hello_world-clean:
        rm -f $(TARGET_DIR)/$(HELLOWORLD_TARGET_BINARY)
        -$(MAKE) -C $(HELLOWORLD_DIR) clean

hello_world-dirclean:
        rm -rf $(HELLOWORLD_DIR)

##############################################################
#
# Toplevel Makefile options
#
##############################################################
ifeq ($(strip $(BR2_PACKAGE_HELLOWORLD)),y)
TARGETS+=hello_world
endif
```

CONCLUSION

L'article nous a démontré qu'il était relativement simple de construire une distribution à l'aide de Buildroot. La version 2009.11 a atteint une maturité permettant de l'envisager sans hésitation comme solution de développement industriel ou pour appréhender sans trop de difficulté les concepts de Linux embarqué.

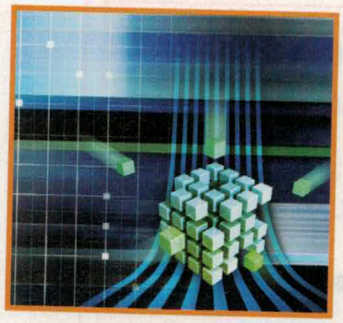
Nous vous donnons rendez-vous dans l'article « Buildroot avancé » pour la suite de nos aventures embarquées dans le monde réel et hostile du véritable matériel :-)

Liens

- Le projet QEMU, **http://bellard.org/qemu**
- Le dépôt Git de QEMU, **http://git.savannah.gnu.org/cgit/qemu.git**
- Le projet COUVERTURE, qui utilise QEMU comme outil de couverture de code embarqué, **http://www.projet-couverture.com**
- ELDK-4.1, **http://ftp.denx.de/pub/eldk/4.1**
- Projet Busybox, **http://www.busybox.net**
- Utilitaire genext2fs, **http://genext2fs.sourceforge.net**
- Le script mklibs (utilisé dans l'article), **http://pficheux.free.fr/articles/lmf/advanced_qemu/mklibs**
- Paquetage mklibs Debian, **http://packages.debian.org/unstable/devel/mklibs**
- Démonstration QEMU/ARM9/DirectFB par Free Electrons, **http://free-electrons.com/community/demos/qemu-arm-directfb**
- Projet Buildroot, **http://buildroot.uclibc.org**
- Miroir des archives sources utilisées par Buildroot, **http://buildroot.uclibc.org/downloads/sources**
- Documentation Buildroot, **http://buildroot.uclibc.org/buildroot.html**
- Présentation de Thomas Petazzoni aux RMLL 2009, **http://2009.rmll.info/IMG/pdf**
- Outil Crosstool, **http://www.kegel.com/crosstool**
- Outil Crosstool-NG, **http://ymorin.is-a-geek.org/dokuwiki/projects/crosstool**

Auteur : Pierre Ficheux

Cas pratique d'utilisation



Dans l'article d'introduction à Buildroot, nous avons présenté les principaux concepts de cet outil. Cependant, il est très fréquent que l'utilisateur doive adapter l'environnement de développement à son matériel, souvent spécifique.

Auteur

- Pierre Ficheux

Dans ce deuxième article, un test sera effectué sur une carte réelle à base de processeur ARM9 (SAMSUNG S3C2410) afin d'expliciter la procédure d'ajout d'une nouvelle architecture matérielle à Buildroot, puis le test du résultat dans l'environnement du bootloader U-Boot.

Nous évoquerons également l'utilisation d'une chaîne de compilation externe basée sur Glibc.

Les exemples présentés sont disponibles sur **http://pficheux.free.fr/articles/lmf/buildroot/exemples_adv.tgz**.

1 UTILISATION DE BUILDROOT SUR UNE VÉRITABLE CARTE

Dans l'article d'introduction, nous avons testé Buildroot dans l'environnement émulé QEMU. Dans la suite, nous allons mettre en place une distribution complète pour une carte basée sur un processeur ARM9 de chez SAMSUNG (S3C2410). La carte est le modèle DEV2410 de la société Pragmatec (**http://www.pragmatec.net**). Pragmatec commercialise également une carte très proche sous forme de module (SODIMM2410), basée sur le même processeur.

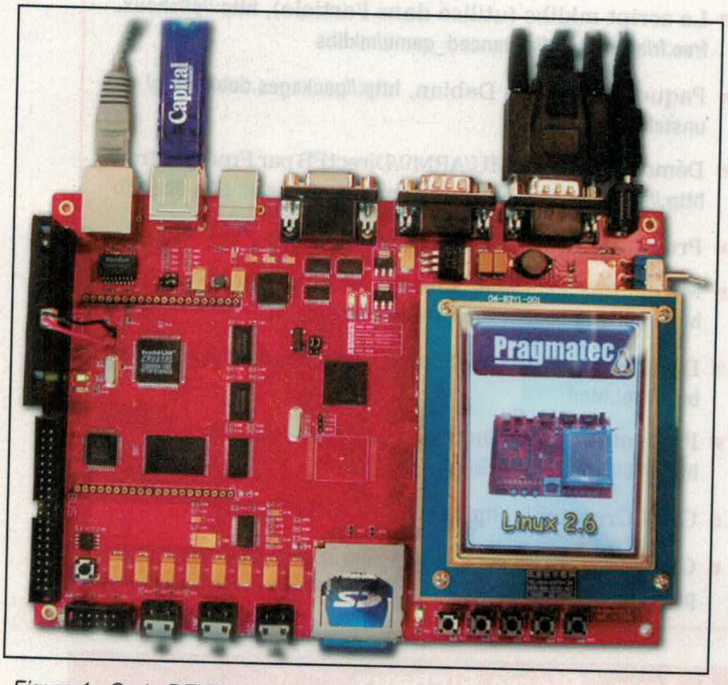


Figure 1 : Carte DEV2410 (source : Pragmatec)

Cette carte peut supporter le *bootloader* U-Boot même s'il n'est pas installé par défaut. Elle dispose de 64 Mo de flash NAND, 64 Mo de SDRAM ainsi que différents périphériques classiques visibles sur la photo : Ethernet 100 Mbps, USB *host* et *device*, écran graphique, lecteur de carte SD.

Le module SODIMM2410 est visible ci-contre.

La société Pragmatec fournit une distribution « maison » basée sur un noyau 2.6.28.7 adapté, mais cette distribution n'utilise pas d'outil type Buildroot. Dans la suite de l'article, nous allons modifier la version 2009.11 de Buildroot afin de supporter les cartes DEV2410 et SODIMM2410.

Nous pourrions bien entendu réaliser un test similaire à celui effectué avec QEMU, soit :

- Générer une chaîne de compilation croisée avec Buildroot (basée sur uClibc) ou utiliser une chaîne externe produite, par exemple, avec Crosstool.
- Générer une image du root-filesystem avec Buildroot.
- Compiler le noyau 2.6.28.7 en dehors de Buildroot avec la même chaîne de compilation.

Cependant, cette solution n'utilise pas au maximum les possibilités de Buildroot. En effet, une fois bien adapté, Buildroot permet de générer l'intégralité des composants utilisés sur la carte avec une seule session de configuration/compilation. Le résultat de la session est décrit ci-dessous :

- une chaîne de compilation basée sur uClibc ;
- le bootloader U-Boot adapté utilisant les patchs du constructeur ;
- le noyau Linux 2.6.28.7 adapté de la même manière ;
- le root-filesystem sous différents formats.

de Buildroot

La carte n'étant pas prise en compte par Buildroot, nous devrons tout d'abord effectuer des modifications proches de celles décrites pour l'ajout d'un paquetage.

Concernant le root-filesystem, nous avons pour l'instant utilisé une image **INITRD** (format CPIO), car ce format était le mieux adapté au test QEMU. Dans le cas d'une carte réelle, on pourra utiliser un root-filesystem monté via le protocole NFS ou une image JFFS2 à flasher sur la carte.

Concernant le noyau Linux, on pourra dans un premier temps le charger en utilisant le protocole TFTP avant de le flasher sur la carte.

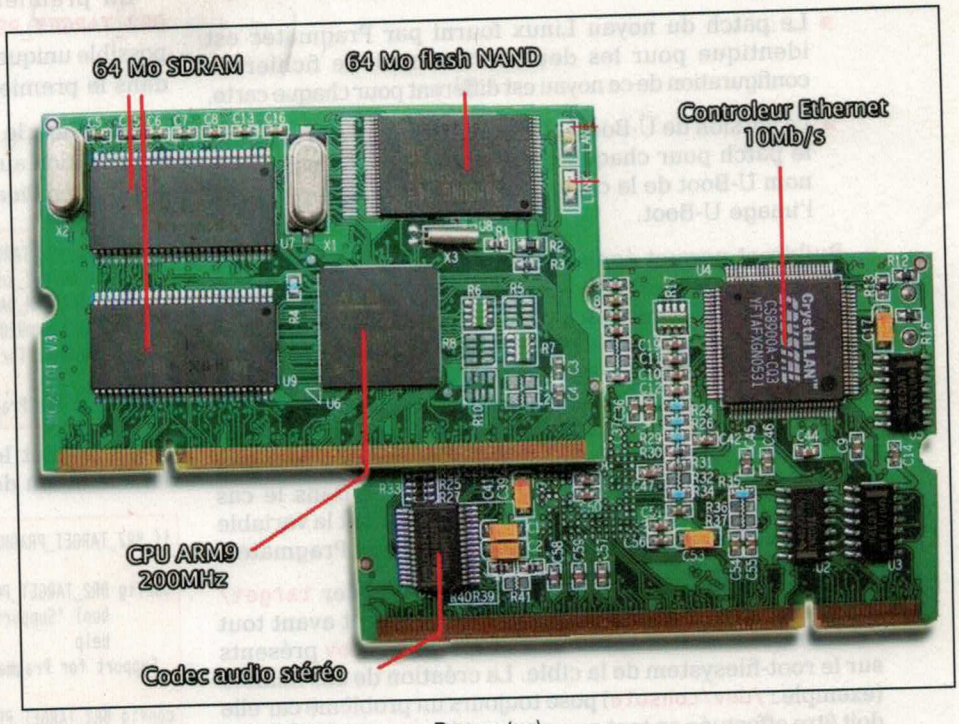


Figure 2 : Module SODIMM2410 (source : Pragmatec)

1.1 Ajout du support des cartes Pragmatec DEV2410 et SODIMM2410

La sélection d'une carte donnée est accessible dans Buildroot par le menu *Target options* reproduit ci-après. Dans le cas des processeurs ARM (*Target Architecture* étant positionné sur arm), la version 2009.11 inclut un certain nombre de cartes : ARM Ltd, Atmel, KwickByte, etc. Bien entendu, la liste affichée dépend du type d'architecture choisi, soit *Target Architecture* et *Target Architecture Variant*.

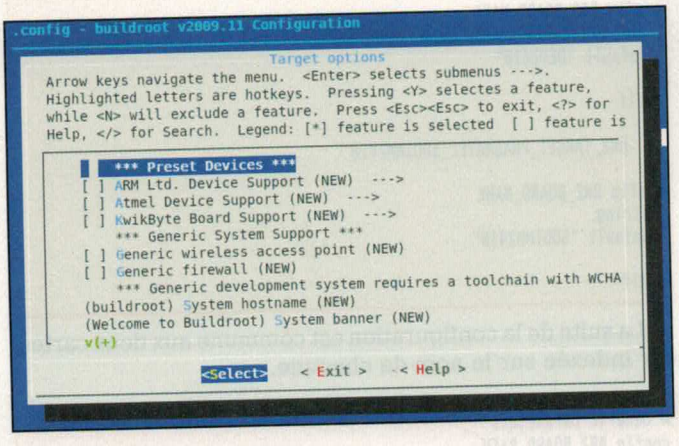


Figure 3 : Options de la cible

1.1.1 Ajout du support Pragmatec au menu Target options

La liste des fichiers décrivant les cibles prédéfinies (*Preset Devices*) est située dans le répertoire **target/device**.

```
$ ls -l target/device
total 60
drwxr-xr-x  4 pierre users 4096 déc.   1 15:27 ARMLTD
drwxr-xr-x 20 pierre users 4096 déc.   1 15:27 Atmel
-rw-r--r--  1 pierre users  350 déc.   1 15:27 Config.in
-rw-r--r--  1 pierre users  300 déc.   1 15:27 Config.in.linux.patches
-rw-r--r--  1 pierre users 2323 déc.   1 15:27 Config.in.mirrors
-rw-r--r--  1 pierre users 5703 déc.   1 15:27 Config.in.toolchain
-rw-r--r--  1 pierre users  259 déc.   1 15:27 Config.in.u-boot
drwxr-xr-x  3 pierre users 4096 déc.   1 15:27 KwikByte
-rw-r--r--  1 pierre users  143 déc.   1 15:27 Makefile.in
-rw-r--r--  1 pierre users   46 déc.   1 15:27 Makefile.in.linux
drwxr-xr-x  2 pierre users 4096 déc.   1 15:27 mips
drwxr-xr-x  4 pierre users 4096 déc.   1 15:27 valka
drwxr-xr-x  3 pierre users 4096 déc.   1 15:27 x86
drwxr-xr-x  3 pierre users 4096 déc.   1 15:27 xtensa
```

Comme pour la définition des paquetages, l'ajout d'un support de carte passe par la modification de plusieurs fichiers :

- Ajout d'une ligne dans **target/device/Config.in**

```
source "target/device/Pragmatec/Config.in"
```

- Création d'un répertoire **target/device/Pragmatec** (nom du constructeur). Ce répertoire contient lui-même un répertoire par nouvelle carte supportée (DEV2410 et SODIMM2410 dans notre cas).
- Le répertoire **Pragmatec** contient le fichier **Config.in** permettant de sélectionner la carte dans la liste. Il contient également le fichier **Makefile.in** définissant les variables utilisées pour la compilation. Chaque répertoire de carte, comme **DEV2410**, peut également contenir un fichier **Makefile.in** si nécessaire.

La meilleure solution est de décrire le contenu des fichiers déduit de l'architecture des cartes Pragmatec.

- Les cartes dont nous disposons utilisent le même CPU S3C2410, qui contient un cœur ARM920T.
- Le patch du noyau Linux fourni par Pragmatec est identique pour les deux cartes, mais le fichier de configuration de ce noyau est différent pour chaque carte.
- La version de U-Boot utilisée est la même (1.1.6), mais le patch pour chaque carte est différent ainsi que le nom U-Boot de la carte utilisée pour la génération de l'image U-Boot.

Buildroot permet également d'adapter le « squelette » du root-filesystem produit pour une carte. Ce squelette est constitué de l'arborescence d'un système Linux embarqué auquel on aurait retiré les fichiers exécutables, les bibliothèques et tous les fichiers binaires en général. Le squelette par défaut fourni par Buildroot est situé dans le répertoire **target/generic/target_skeleton**. Le squelette utilisé peut être modifié dans la configuration de chaque carte en définissant la variable **TARGET_SKELETON**. Dans le cas présent, nous utiliserons le squelette standard et la variable ne sera donc pas définie dans la configuration Pragmatec.

Outre le squelette, Buildroot fournit le fichier **target/generic/device_table.txt**. Ce fichier permet avant tout de définir les fichiers spéciaux du répertoire **/dev** présents sur le root-filesystem de la cible. La création de ces fichiers (exemple : **/dev/console**) pose toujours un problème car elle doit être effectuée en tant que super-utilisateur. Dans l'article d'introduction, nous avions construit ces fichiers manuellement en utilisant la commande **MAKEDEV**, mais il était indispensable d'utiliser la commande **sudo** afin d'être super-utilisateur.

```
$ mkdir dev
$ sudo MAKEDEV -v -d dev generic console
```

Buildroot utilise une technique beaucoup plus efficace, qui évite de passer super-utilisateur. Cette technique est basée sur l'utilisation de la commande **fakeroot**, qui permet de simuler l'utilisation des droits du super-utilisateur pour la manipulation des fichiers. La liste fournie par défaut décrit un répertoire **/dev** utilisable dans la majorité des cas. Cependant, si l'on veut ajouter une entrée spéciale correspondant à un pilote particulier – ou simplement un fichier particulier – on peut définir une nouvelle liste en affectant la variable **TARGET_DEVICE_TABLE**. Nous verrons un exemple d'utilisation dans le fichier **Makefile.in** ci-après.

Le répertoire **target/device/Pragmatec** contient donc les fichiers suivants :

```
$ ls -l target/device/Pragmatec
total 20
-rw-r--r-- 1 pierre users 1035 janv. 24 16:24 Config.in
drwxr-xr-x 2 pierre users 4096 janv. 24 16:23 DEV2410
drwxr-xr-x 2 pierre users 4096 janv. 24 16:23 kernel-patches-2.6.28.7
-rw-r--r-- 1 pierre users  199 janv. 24 16:23 Makefile.in
drwxr-xr-x 2 pierre users 4096 janv. 24 16:23 SODIMM2410
```

Etant donné que les deux cartes utilisent le même patch noyau, le répertoire **kernel-patches-2.6.28.7** est situé au niveau du répertoire du constructeur et non dans les répertoires des cartes, ce qui évite de dupliquer les données.

La majeure partie de la configuration est définie dans le fichier **Config.in**, inclus dans le fichier **target/device/Config.in**.

En premier lieu, on définit une nouvelle variable **BR2_TARGET_PRAGMATEC**. L'affichage de ce nouveau menu est possible uniquement si l'on a sélectionné l'architecture ARM dans le premier écran de Buildroot (*depends on* **BR2_arm**).

De même, le choix de cette nouvelle architecture implique la validation automatique des variables **BR2_ARM_OABI** (*Old Application Binary Interface*) et **BR2_arm920t** (cœur ARM920T).

```
menuconfig BR2_TARGET_PRAGMATEC
  depends on BR2_arm
  select BR2_ARM_OABI
  select BR2_arm920t
  bool "Support for Pragmatec boards"
  help
    Support for Pragmatec boards
```

Il s'en suit le code permettant la sélection des cartes dans le menu de configuration.

```
if BR2_TARGET_PRAGMATEC

config BR2_TARGET_PRAGMATEC_DEV2410
       bool "Support for Pragmatec DEV2410"
       help
  Support for Pragmatec DEV2410

config BR2_TARGET_PRAGMATEC_SODIMM2410
       bool "Support for Pragmatec SODIMM2410"
       help
  Support for Pragmatec SODIMM2410
```

Pour chaque carte, on définit son nom, ce qui permet de placer les paramètres spécifiques dans chaque sous-répertoire **DEV2410** et **SODIMM2410**.

```
if  BR2_TARGET_PRAGMATEC_DEV2410

config BR2_BOARD_NAME
  string
  default "DEV2410"

endif

if  BR2_TARGET_PRAGMATEC_SODIMM2410

config BR2_BOARD_NAME
  string
  default "SODIMM2410"

endif
```

La suite de la configuration est commune aux deux cartes, car indexée sur le nom de chacune.

```
# Generic parameters
config BR2_BOARD_PATH
  string
  default "target/device/Pragmatec/$(BR2_BOARD_NAME)"

# Pragmatec uses specific kernel version
config BR2_KERNEL_ARCH_PATCH_VERSION
  string
  depends on BR2_KERNEL_ARCH_PATCH_ENABLED
```

```
  default "2.6.28.7"

# Kernel patch is in kernel-patches-2.6.28.7 directory
config BR2_KERNEL_ARCH_PATCH_DIR
  string
  default "target/device/Pragmatec/kernel-patches-$(BR2_KERNEL_ARCH_
  PATCH_VERSION)"

# Both board uses U-Boot-1.1.6
config BR2_UBOOT_VERSION
  string
  default "1.1.6"

endif
```

Le fichier **Makefile.in** dédié est assez réduit. On définit la version spéciale du fichier **device_table.txt** ainsi que pour l'exemple, une configuration particulière de Busybox. Chaque fichier est situé sur le répertoire de la carte, car indexé sur **BOARD_PATH**.

```
BOARD_NAME:=$(call qstrip,$(BR2_BOARD_NAME))
BOARD_PATH:=$(call qstrip,$(BR2_BOARD_PATH))

TARGET_DEVICE_TABLE=$(BOARD_PATH)/device_table.txt
BR2_PACKAGE_BUSYBOX_CONFIG=$(BOARD_PATH)/busybox.config
```

Les répertoires des cartes contiennent uniquement des fichiers de données. Le patch appliqué à U-Boot est différent pour chaque carte ainsi que les fichiers de configuration pour le noyau Linux et Busybox.

```
$ ls -l target/device/Pragmatec/*2410*
target/device/Pragmatec/DEV2410:
total 212
-rw-r--r-- 1 pierre users  23727 janv. 24 16:23 busybox.config
-rw-r--r-- 1 pierre users  42359 janv. 24 16:23 DEV2410-linux-2.6.28.7.config
-rw-r--r-- 1 pierre users   6263 janv. 24 16:23 device_table.txt
-rw-r--r-- 1 pierre users 135718 janv. 24 16:23 u-boot-1.1.6.patch

target/device/Pragmatec/SODIMM2410:
total 192
-rw-r--r-- 1 pierre users  23727 janv. 24 16:23 busybox.config
-rw-r--r-- 1 pierre users   6263 janv. 24 16:23 device_table.txt
-rw-r--r-- 1 pierre users  35208 janv. 24 16:23 SODIMM2410-linux-2.6.28.7.config
-rw-r--r-- 1 pierre users 126715 janv. 24 16:23 u-boot-1.1.6.patch
```

Suite à ces diverses modifications, on peut voir apparaître le menu *Support for Pragmatec board* dans *Target options*. Si l'on active le support Pragmatec par la barre d'espace, on peut sélectionner le support DEV2410 ou SODIMM2410.

```
    *** Preset Devices ***
[ ] ARM Ltd. Device Support  --->
[ ] Atmel Device Support  --->
[ ] KwikByte Board Support  --->
[*] Support for Pragmatec boards  --->
    *** Generic System Support ***
[ ] Generic wireless access point
[ ] Generic firewall
```

Figure 4 : Ajout du support Pragmatec à la liste

```
--- Support for Pragmatec boards
[*]   Support for Pragmatec DEV2410
[ ]   Support for Pragmatec SODIMM2410
```

Figure 5 : Sélection de la carte DEV2410

Nous remarquons cependant que la configuration n'est pas totalement terminée, car le nom du système est toujours buildroot et la bannière d'accueil a conservé la valeur par défaut. De même, si l'on se place dans le menu Kernel, on constate qu'il est toujours à la valeur par défaut *none*.

On peut en théorie définir toutes les variables **BR2_*** dans le fichier **Config.in**, comme nous l'avons fait pour **BR2_BOARD_NAME**.

```
config BR2_BOARD_NAME
  string
  default "DEV2410"
```

Cependant, c'est un travail fastidieux car il faut récupérer les noms des nombreuses variables associées. Il est plus simple d'utiliser l'outil de configuration, puis de sauver le fichier **.config** lorsqu'il correspond exactement à la configuration voulue. Dans le cas des nouvelles cartes, il reste à configurer les points suivants :

- chaîne de compilation (menu *Toolchain*). Pour ce point, nous utiliserons la configuration de chaîne uClibc proposée par défaut par Buildroot. Ce point ne sera donc pas détaillé ;
- bootloader U-Boot (menu *Target filesystem options*) ;
- noyau Linux (menu *Kernel*) ;
- format des images de root-filesystem générées (menu *Target filesystem options*) ;
- Périphérique (UART) utilisé pour la console (menu *Target options*).

1.1.2 Configuration U-Boot

Concernant U-Boot, on accède à la configuration par la dernière ligne du menu Target filesystem options, intitulée très justement de par les origines germaniques de U-Boot « *Das U-Boot Boot Monitor* ».

```
--- Das U-Boot Boot Monitor
(dev2410) board name
      U-Boot Version (u-boot-2009.08)  --->
[ ]   Add architecture specific patch  --->
($(BR2_BOARD_PATH)/u-boot-1.1.6.patch) custom patch
```

Figure 6 : Configuration U-Boot pour la carte DEV2410

Le chemin d'accès au patch à appliquer est donné par l'option *custom patch*, qui correspond à la variable **CONFIG_BR2_TARGET_UBOOT_CUSTOM_PATCH** dans le fichier **.config**. La version de U-Boot affichée (u-boot-2009.08) n'est pas utilisée puisque nous avons forcé la variable **BR2_UBOOT_VERSION** à **1.1.6** dans le fichier **target/device/Pragmatec/Config.in**.

On remarquera le nom dev2410 au lieu de DEV2410. En effet, ce nom correspond à la configuration U-Boot, c'est-à-dire au nom de la cible à utiliser pour la compilation du bootloader. Dans le cas du patch fourni par Pragmatec, la cible est dev2410 et en cas de compilation manuelle, la sélection de la cible s'effectuerait par la commande :

```
$ make dev2410_config
```

1.1.3 Configuration du noyau Linux

Pour définir la configuration du noyau, on utilise le dernier menu de l'outil de configuration Buildroot, intitulé Kernel. Par défaut, Buildroot est configuré pour ne pas traiter le noyau Linux : *Kernel type (none)*.

```
(X) none
( ) Linux (Advanced configuration)
( ) Linux (Same version as linux headers)
```

Figure 7 : Configuration du noyau par défaut

Dans le cas de la carte Pragmatec, le noyau est une version adaptée du noyau 2.6.28.7. On doit donc choisir l'option *Advanced configuration*, qui permet de préciser tous les paramètres du noyau compilé : version, nom du fichier de patch, format, fichier de configuration.

```
    Kernel type (linux (Advanced configuration))  --->
    Linux Kernel Version (Linux <custom> version)  --->
(2.6.28.7) Linux Tarball version
(2.6.28.7) Linux Version
()  patch name
()  patch site
    Patches  --->
    Linux Kernel Configuration  --->
    Kernel binary format (uImage)  --->
    Destinations for linux kernel binaries  --->
```

Figure 8 : Configuration du noyau Pragmatec

On note que le nom du noyau produit est **uImage**, puisque la carte utilise U-Boot. Le fichier **zImage** sera traité par la commande **mkimage** pour produire le fichier **uImage** compatible avec U-Boot.

Si l'on sélectionne l'option *Patches*, on affiche le menu suivant dans lequel on coche l'option *Add ARCH specific patch*.

```
[*] Add ARCH specific patch  --->
()  Additional patch to apply (from $(DL_DIR))
```

Figure 9 : Sélection du type de patch noyau

Cela signifie que l'on utilisera la variable **BR2_KERNEL_ARCH_PATCH_DIR** pour obtenir le nom du fichier de patch. Dans notre cas, la variable a la valeur :

```
target/device/Pragmatec/$(BR2_BOARD_NAME)/kernel-patches-$(BR2_KERNEL_
ARCH_PATCH_VERSION)
```

Cette valeur est donc indexée par le nom de la carte et la version spéciale du noyau.

De même, le chemin d'accès au fichier de configuration du noyau dépend du nom de la carte et de la version du noyau.

```
    Linux Kernel Configuration (.config file)  --->
($(BR2_BOARD_PATH)/$(BR2_BOARD_NAME)-linux-$(BR2_LINUX26_VERSION)
```

Figure 10 : Sélection du fichier de configuration

1.1.4 Format des images de root-filesystem produites

Lors de la compilation, une « copie » du root-filesystem de la cible est produite sur le répertoire **output/target**. Ce n'est pas une véritable copie, car le répertoire **output/target/dev** ne contient pas les fichiers spéciaux décrits dans **device_table.txt**, mais des fichiers standards de taille nulle portant les mêmes noms. Les véritables entrées dans **/dev** sont uniquement présentes dans les images produites dans **output/images** créées à l'aide de l'utilitaire **fakeroot** déjà cité.

Le menu Target filesystems options a déjà été évoqué lors de la construction de l'image testée avec QEMU (voir figure 9 de l'article d'introduction). Dans le cas de QEMU, nous avions produit une image au format CPIO utilisable pour un *ramdisk*.

Buildroot est capable de produire un grand nombre d'autres formats : cramfs, cloop, ext2, jffs2, squashfs, tar, initramfs, romfs.

Dans le cas de la carte DEV2410, nous produirons une image tar et une image jffs2.

- L'image tar servira à créer un répertoire contenant le root-filesystem sur le poste de développement. Ce répertoire sera utilisé par la carte via NFS en utilisant la technique dite du « NFS-Root ». Cette méthode est très utilisée lors de la phase de développement, car le contenu du root-filesystem peut être facilement modifié sur le poste de développement sans redémarrer la carte.
- L'image jffs2 correspond à une image à flasher sur la carte DEV2410. Le format de l'image produite dépend de la géométrie de la mémoire flash. Ces paramètres peuvent être définis directement dans le menu Buildroot et ils seront passés à l'utilitaire **mkfs.jffs2**.

Dans les deux cas, nous décrirons brièvement l'utilisation des images tar et jffs2 dans U-Boot.

Dans le cas de l'image tar, on précise uniquement le type de compression gzip.

```
[*] tar the root filesystem
    Compression method (gzip)  --->
()  other random options to pass to tar
()  also copy the image to...
```

Figure 11 : Paramètres de l'image au format tar

Dans le cas de l'image jffs2, les choses sont plus compliquées car l'image doit correspondre aux paramètres physiques de la flash, en particulier la taille de page (*page size*) et la taille d'effacement (*erase block size*). Si les paramètres sont incorrects, le root-filesystem produit pourra être inutilisable !

```
[ ] ext2 root filesystem
[*] jffs2 root filesystem
    Flash Type (Select custom page and erase size)  --->
(0x1000) Page Size
(0x4000) Erase block size
[*]   Do not use Cleanmarker
[ ]   RootFS in SREC file formet
[*]   Pad output
(0x0)   Pad output size (0x0 = to end of EB)
    Endianess (little-endian)  --->
[ ]   Produce a summarized JFFS2 image
($(IMAGE).jffs2) Output File
()    also copy the image to...
```

Figure 12 : Paramètres de l'image au format jffs2

Les paramètres sont disponibles dans la documentation de la carte ou de la mémoire flash. L'erase block size a souvent une valeur de 128 KiB (valeur par défaut 0x20000), mais dans notre cas, elle est de 16 KiB (0x4000).

1.1.5 Périphérique utilisé pour la console

Nous avons évoqué le sujet lors de la configuration Buildroot pour QEMU. Les noms du système (*hostname*) et la bannière d'accueil ne posent pas vraiment de problème. Par contre, une mauvaise configuration du paramètre *Generic serial port config* peut conduire à une image inutilisable. En effet, dans le cas de QEMU, ce dernier émulait le framebuffer de la carte et la console utilisait donc le périphérique **/dev/tty1**, soit une console similaire à celle d'un PC/x86.

Dans le cas de la carte DEV2410, la console est connectée à un port série physique (UART) dont le nom est **/dev/ttySAC0** (et proposé par Buildroot dans une liste dépendante de l'architecture utilisée). Si cette option n'est pas précisée, on n'obtiendra jamais la bannière de *login* !

```
(dev2410) System hostname
(Welcome to Pragmatec DEV2410 (uclibc)) System banner
[*] Generic serial port config  --->
```

Figure 13 : Nom du système et bannière

```
--- Generic serial port config
    Serial port to run a getty on (ttySAC0)  --->
    Baudrate to use (115200)  --->
```

Figure 14 : Définition du port série de la console

1.1.6 Cas d'une chaîne de compilation externe

La configuration par défaut utilise un compilateur basé sur uClibc et produit par Buildroot. Il peut être nécessaire d'utiliser un autre compilateur pour diverses raisons :

- incompatibilité de bibliothèques applicatives tierces avec uClibc (exemple : extensions temps réel) ;
- choix technique ou historique pour les développements ;
- utilisation d'un outil commercial comme CodeSourcery.

Dans ces cas-là, il conviendra de configurer Buildroot afin d'utiliser une chaîne de compilation « externe ». Le menu Toolchain peut être utilisé comme suit :

```
    Toolchain type (External binary toolchain)  --->
    External toolchain C library (glibc)  --->
[*] Strip shared libraries
    *** Gdb Options ***
[ ] Build gdb debugger for the Target
[ ] Build gdb server for the Target
[ ] Build gdb for the Host
    *** Common Toolchain Options ***
[*] Enable large file (files > 2 GB) support?
[*] Enable IPv6
[*] Enable RPC
[*] Enable toolchain locale/i18n support?
[ ] Purge unwanted locales
-*- Enable WCHAR support
[*] Use software floating point by default
[ ] Enable stack protection support
    Thread library implementation (linuxthreads (stable/old))  --->
[*] Enable 'program invocation name'
[ ] Build/install c++ compiler and libstdc++?
(-Os -pipe) Target Optimizations
(${HOME}/crosstool/gcc-4.1.0-glibc-2.3.2/arm-unknown-linux-gnu) External toolchain path
```

Figure 15 : Définition d'une chaîne externe

Dans notre cas, la chaîne est située sur **$HOME/crosstool/gcc-4.1.0-glibc-2.3.2**. Le nom du compilateur est **arm-unknown-linux-gnu-gcc**.

La méthode de production de la chaîne dépasse le sujet de l'article. On pourra se référer à l'article « Compilation croisée sous Linux et Windows » du même auteur ou consulter des outils comme Crosstool ou Crosstool-NG cités en bibliographie.

Remarque

L'utilisation d'une chaîne externe sous Buildroot 2009.11 nécessite que celle-ci accepte l'option **--sysroot**. Cette option permet de spécifier le chemin d'accès des bibliothèques et des en-têtes de la chaîne de compilation, elle est disponible uniquement sur les versions 4 de GCC (de ce fait, la chaîne fournie par Pragmatec n'est pas utilisable en tant que chaîne externe).

1.1.7 Sauvegarde de la configuration

Lorsque la configuration est terminée, on doit sauver le fichier **.config**, comme dans le cas de QEMU.

Le fichier **.config** pourra ensuite être copié dans le répertoire **configs**, qui contient les configurations prédéfinies des cartes supportées par Buildroot.

```
$ ls -l configs
total 760
-rw-r--r-- 1 pierre users 18897 déc.   1 15:27 arm_toolchain_defconfig
-rw-r--r-- 1 pierre users 28256 déc.   1 15:27 at91rm9200df_defconfig
-rw-r--r-- 1 pierre users 16731 déc.   1 15:27 at91rm9200df_ext_bare_defconfig
…
-rw-r--r-- 1 pierre users 20495 janv. 24 16:23 pragmatec_DEV2410_glibc_defconfig
-rw-r--r-- 1 pierre users 23017 janv. 24 16:23 pragmatec_DEV2410_uclibc_defconfig
-rw-r--r-- 1 pierre users 20560 janv. 24 16:23 pragmatec_SODIMM2410_glibc_defconfig
-rw-r--r-- 1 pierre users 23027 janv. 24 16:23 pragmatec_SODIMM2410_uclibc_defconfig
-rw-r--r-- 1 pierre users 19931 déc.   1 15:27 v100sc2_defconfig
```

Dans le cas présent, nous avons défini quatre fichiers de configuration dédiés aux nouvelles cartes. Pour chaque carte, nous avons un fichier utilisant un compilateur Buildroot basé sur uClibc et un fichier utilisant un compilateur externe basé sur Glibc. Pour utiliser une configuration puis compiler le support pour une carte, il suffira donc de taper **uniquement** les deux lignes suivantes :

```
$ make pragmatec_DEV2410_uclibc_defconfig
$ make
```

2 TEST SUR LA CARTE DEV2410

A l'issue de l'exécution des deux commandes précédentes, les images sont disponibles dans le répertoire **output/images**.

```
$ ls -l output/images/
total 7016
-rw-r--r-- 1 pierre users 1540096 janv. 23 22:50 rootfs.arm.jffs2
-rw-r--r-- 1 pierre users 2979840 janv. 23 22:50 rootfs.arm.tar
-rwxr-xr-x 1 pierre users  100284 janv. 20 18:46 u-boot-1.1.6-20100120.
 bin
lrwxrwxrwx 1 pierre users      25 janv. 20 18:55 u-boot.bin ->
u-boot-1.1.6-20100120.bin
-rw-r--r-- 1 pierre users 2559516 janv. 20 18:55 uImage
```

Pour être utilisables, les images du noyau et du root-filesystem devront être placées sur le répertoire du serveur TFTP du poste de développement, en général **/tftpboot** ou **/var/lib/tftpboot**.

```
$ sudo cp uImage /tftpboot/uImage_prag_br
$ sudo cp rootfs.arm.jffs2 /tftpboot
```

L'image U-Boot doit être flashée sur la carte grâce à la sonde JTAG fournie par le constructeur. On peut également flasher U-Boot à partir du bootloader Pragmatec en utilisant un simple lien série (voir la documentation du constructeur).

Nous utilisons l'émulateur de terminal **minicom** disponible sur toutes les distributions Linux. La carte a été testée en mode NFS-Root, puis en flashant les images noyau et root-filesystem par le bootloader U-Boot. Lors du démarrage de la carte, on doit obtenir la bannière U-Boot ci-dessous :

```
 ********************************************************
*   U-Boot 1.1.6 (Sep  7 2009 - 14:34:35)               *
*                                                       *
*                                  www.pragmatec.net *
* PRAGMATEC                        www.pragmatux.net *
 ********************************************************
DRAM:  64 MB
NAND:  64 MB
progress 100%
In:    serial
Out:   serial
Err:   serial
CS8900 detected...
Hit any key to stop autoboot:  0
S3C2410 #
```

Une description détaillée du fonctionnement de U-boot dépasse le cadre de cet article, mais nous pouvons donner quelques informations indispensables à son utilisation. Une description détaillée est disponible chez DENX Software, qui maintient U-Boot (http://www.denx.de/wiki/U-Boot).

U-Boot utilise des variables d'environnement « à la UNIX ». On peut créer de nouvelles variables, mais certaines variables prédéfinies sont indispensables au fonctionnement. Une variable peut également être une macro qui exécute une suite de commandes.

La liste des principales commandes est la suivante :

Nom de la commande	Description
printenv	Affichage du contenu des variables
setenv	Affectation d'une variable
saveenv	Sauvegarde des variables sur la flash
tftp ou **tftpboot**	Chargement d'un fichier par TFTP
boot	Exécution de la macro **bootcmd**
bootm	Démarrage à une adresse RAM
nboot	Chargement de code de flash NAND vers la RAM
bdinfo	Affiche des informations sur la carte
nand erase	Effacement flash NAND
nand write	Ecriture de données de RAM vers la flash NAND
run	Exécution d'une macro

La liste des principales variables est la suivante :

Nom de la commande	Description
bootargs	Paramètres du noyau Linux
ipaddr	Adresse IP de la carte
serverip	Adresse IP du serveur TFTP
gatewayip	Adresse IP de la passerelle d'accès par défaut
bootfile	Fichier par défaut chargé par la commande **tftp**
bootcmd	Macro appelée par la commande **boot**
loadaddr	Adresse de chargement en RAM par TFTP
ethaddr	Adresse MAC de la carte
filesize	Taille du dernier fichier chargé par TFTP
rootpath	Répertoire du root-filesystem NFS sur le poste de développement

Dans le cas de la carte DEV2410, nous avons défini un certain nombre de variables et macros supplémentaires afin de faciliter l'utilisation. On peut les afficher par la commande **printenv**.

```
S3C2410 # printenv
baudrate=115200
ethaddr=08:00:3e:26:0a:5b
loadaddr=33000000
bootcmd=tftpboot ${loadaddr} ${bootfile}; bootm ${loadaddr}
bootdelay=5
nfsargs=setenv bootargs console=ttySAC0,115200 root=/dev/nfs
nfsroot=${serverip}\
:${rootpath} ip=${ipaddr}:${serverip}:${gatewayip}:${netmask}:::off
```

```
nandargs=setenv bootargs console=ttySAC0,115200
root=/dev/mtdblock2 rootfstype=jffs2
rootpath=/home/pierre/rootfs_br_prag
kern_erase=nand erase 0x30000 0x300000
root_erase=nand erase 0x330000 0x500000
kern_flash=nand write ${loadaddr} 0x30000 ${filesize}
root_flash=nand write ${loadaddr} 0x330000 ${filesize}
filesize=178000
fileaddr=33000000
netmask=255.255.255.0
ipaddr=192.168.3.50
serverip=192.168.3.109
kern_install=run kern_erase; tftp ${loadaddr} ${bootfile} ; run kern_
flash
rootfile=rootfs.arm.jffs2
root_install=run root_erase ; tftp ${loadaddr} ${rootfile} ; run root_flash
nand_boot=run nandargs ; nboot 33000000 0 30000; bootm
nfs_boot=run nfsargs; tftpboot ${loadaddr} ${bootfile}; bootm ${loadaddr}
bootargs=console=ttySAC0,115200 root=/dev/nfs nfsroot=192.168.3.109:/
home/pierre \
/rootfs_br_prag ip=192.168.3.50:192.168.3.109::255.255.255.0:::off
bootfile=uImage_prag_br
stdin=serial
stdout=serial
stderr=serial
```

Nom de la commande	Description
`nfsargs`	Affecte la valeur de **bootargs** pour le test NFS-Root
`nandargs`	Affecte la valeur de **bootargs** pour le test flash NAND
`nfs_boot`	Démarre la carte en mode NFS-Root, le noyau Linux est chargé par TFTP
`nand_boot`	Démarre la carte depuis la flash NAND (noyau Linux et root-filesystem)
`kern_erase`	Efface la zone de flash réservée au noyau
`root_erase`	Efface la zone de flash réservée au root-filesystem
`kernel_install`	Installe le noyau sur la flash NAND
`root_install`	Installe le root-filesystem sur la flash NAND

2.1 Test en NFS-Root

Pour ce faire, il faut extraire l'image **.tar** (ou **.tar.gz**) sur un répertoire dédié, par exemple **$HOME/rootfs_br_prag**.

```
$ mkdir $HOME/rootfs_br_prag
$ cd $HOME/rootfs_br_prag
$ sudo tar xvf <Buildroot_path>/output/images/rootfs.arm.tar
```

Remarque

Il est nécessaire d'utiliser **sudo** pour créer le contenu du répertoire **/dev** de la cible, car les entrées doivent être créées en tant que super-utilisateur. Côté U-Boot, on utilise la macro **nfs_root** pour lancer le test.

```
S3C2410 # run nfs_boot
TFTP from server 192.168.3.109; our IP address is 192.168.3.50
Filename 'uImage_prag_br'.
Load address: 0x33000000
Loading: #################################################################
...
done
Bytes transferred = 2559516 (270e1c hex)
## Booting image at 33000000 ...
   Image Name:   Linux-2.6.282.6.28.7
   Created:      2010-01-20  17:55:48 UTC
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    2559452 Bytes =  2.4 MB
   Load Address: 30008000
   Entry Point:  30008000
   Verifying Checksum ... OK
OK
Starting kernel ...
Uncompressing Linux.....................
.........................................
..... done, booting the kernel.
Linux version 2.6.282.6.28.7 (pierre@opti760pf.localdomain)
(gcc version 4.3.4 ( GCC) ) #1 Wed Jan 20 18:55:40 CET 2010
CPU: ARM920T [41129200] revision 0 (ARMv4T), cr=c0007177
CPU: VIVT data cache, VIVT instruction cache
Machine: DEV2410 PRAGMATEC
Memory policy: ECC disabled, Data cache writeback
CPU S3C2410A (id 0x32410002)
S3C2410: core 202.800 MHz, memory 101.400 MHz, peripheral 50.700 MHz
S3C24XX Clocks, (c) 2004 Simtec Electronics
CLOCK: Slow mode (1.500 MHz), fast, MPLL on, UPLL on
Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 16256
Kernel command line: console=ttySAC0,115200 root=/dev/nfs
nfsroot=192.168.3.109:
/home/pierre/rootfs_br_prag ip=192.168.3.50:192.168.3.109::255.255.255.0:::off
irq: clearing subpending status 00000002
PID hash table entries: 256 (order: 8, 1024 bytes)
timer tcon=00500000, tcnt a509, tcfg 00000200,00000000, usec 00001e4c
Console: colour dummy device 80x30
console [ttySAC0] enabled
Dentry cache hash table entries: 8192 (order: 3, 32768 bytes)
Inode-cache hash table entries: 4096 (order: 2, 16384 bytes)
Memory: 64MB = 64MB total
Memory: 59500KB available (4752K code, 398K data, 224K init)
Calibrating delay loop... 100.96 BogoMIPS (lpj=252416)
Mount-cache hash table entries: 512
CPU: Testing write buffer coherency: ok
net_namespace: 484 bytes
...
Initializing random number generator... done.
Starting network...
ip: RTNETLINK answers: File exists
Welcome to Pragmatec DEV2410 (uclibc)
dev2410 login: root
# uname -a
Linux dev2410 2.6.282.6.28.7 #1 Wed Jan 20 18:55:40 CET 2010 armv4tl GNU/Linux
#
```

2.2 Test avec la mémoire flash NAND

Le test est similaire, à part que l'on flashe le noyau et l'image root-filesystem auparavant. Pour le noyau, on utilise la macro **kern_install**.

```
S3C2410 # run kern_install
NAND erase: device 0 offset 196608, size 3145728 ...
progress 100%
3145728 bytes sould be erased, 3145728 really erased
OK
```

```
TFTP from server 192.168.3.109; our IP address is 192.168.3.50
Filename 'uImage_prag_br'.
Load address: 0x33000000
Loading: ###################################################################
     ...
         ##################################
Bytes transferred = 2559516 (270e1c hex)
NAND write: device 0 offset 196608, size 2559516 ...
progress 100%
 2559516 bytes written: OK
```

Pour le root-filesystem, on utilise la macro **root_install**.

```
S3C2410 # run root_install
NAND erase: device 0 offset 3342336, size 5242880 ...
progress 100%
5242880 bytes sould be erased, 5242880 really erased
OK
TFTP from server 192.168.3.109; our IP address is 192.168.3.50
Filename 'rootfs.arm.jffs2'.
Load address: 0x33000000
Loading: ###################################################################
     ...
         ##################################
done
Bytes transferred = 1540096 (178000 hex)
NAND write: device 0 offset 3342336, size 1540096 ...
progress 100%
 1540096 bytes written: OK
```

On démarre la carte par la macro **nand_boot**. On remarque le chargement de la flash NAND vers la RAM, puis le démarrage du noyau depuis la RAM, car il n'est pas possible de démarrer directement depuis la flash NAND.

```
S3C2410 # run nand_boot
Loading from device 0: <NULL> at 0x4e000000 (offset 0x30000)
progress 100%
   Image Name:   Linux-2.6.282.6.28.7
   Created:      2010-01-20  17:55:48 UTC
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    2559452 Bytes =  2.4 MB
   Load Address: 30008000
   Entry Point:  30008000
progress 100%
## Booting image at 33000000 ...
   Image Name:   Linux-2.6.282.6.28.7
   Created:      2010-01-20  17:55:48 UTC
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    2559452 Bytes =  2.4 MB
   Load Address: 30008000
   Entry Point:  30008000
   Verifying Checksum ... OK
OK
Starting kernel ...
Uncompressing Linux.....................
..................... done, booting the kernel.
Linux version 2.6.282.6.28.7 (pierre@opti760pf.localdomain)
(gcc version 4.3.4 ( GCC) ) #1 Wed Jan 20 18:55:40 CET 2010
CPU: ARM920T [41129200] revision 0 (ARMv4T), cr=c0007177
CPU: VIVT data cache, VIVT instruction cache
Machine: DEV2410 PRAGMATEC
...
Starting network...
ip: RTNETLINK answers: File exists
Welcome to Pragmatec DEV2410 (uclibc)
dev2410 login: root
```

On note bien l'utilisation du roof-filesystem sur la flash.

```
# mount
rootfs on / type rootfs (rw)
/dev/root on / type jffs2 (rw)
proc on /proc type proc (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
tmpfs on /tmp type tmpfs (rw)
sysfs on /sys type sysfs (rw)
# cat /proc/mtd
dev:    size   erasesize  name
mtd0: 00030000 00004000 "bootloader"
mtd1: 00300000 00004000 "Kernel"
mtd2: 00500000 00004000 "rootfs"
mtd3: 03700000 00004000 "userland"
```

CONCLUSION

Ce deuxième article montre que l'ajout d'une carte dédiée n'est pas trop complexe, à partir du moment où elle est basée sur une architecture matérielle déjà prévue par Buildroot, ce qui représente déjà une bonne partie des choix possibles (x86, ARM, PPC, SH4, ...).

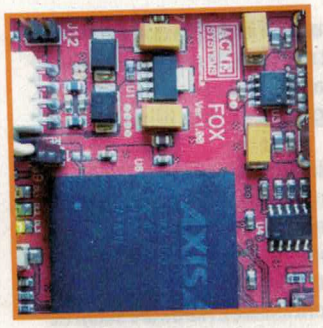
L'adaptation de Buildroot nécessite cependant une bonne connaissance de concepts standards de Linux, comme la structure du fichier **Makefile** ou le script shell.

Liens

- Le script mklibs (utilisé dans l'article), http://pficheux.free.fr/articles/lmf/advanced_qemu/mklibs
- Paquetage mklibs Debian, http://packages.debian.org/unstable/devel/mklibs
- Démonstration QEMU/ARM9/DirectFB par Free Electrons, http://free-electrons.com/community/demos/qemu-arm-directfb
- Projet Buildroot, http://buildroot.uclibc.org
- Miroir des archives sources utilisées par Buildroot, http://buildroot.uclibc.org/downloads/sources
- Documentation Buildroot, http://buildroot.uclibc.org/buildroot.html
- Présentation de Thomas Petazzoni aux RMLL 2009, http://2009.rmll.info/IMG/pdf
- Outil Crosstool, http://www.kegel.com/crosstool
- Outil Crosstool-NG, http://ymorin.is-a-geek.org/dokuwiki/projects/crosstool
- Carte Pragmatec DEV2410, http://www.pragmatec.net/Produits/ARM9.htm
- Forum Pragmatec (annonces, etc.), http://www.pragmatux.net
- Plaquettes des produits, http://www.pragmatec.net/Docs
- Téléchargements, http://www.pragmatec.net/Download
- Projet MTD (*Memory Technology Devices*), http://www.linux-mtd.infradead.org

Auteur : Pierre Ficheux

OpenWrt sur ACME Fox



Auteur

- Denis Bodor

La carte ACME Fox Classic est l'une des premières générations de cartes à base de processeurs ETRAX d'Axis Communications. Il s'agit en réalité d'un SoC ou, comme on l'appelle chez Axis, d'un MCM (Multi Chip Module). Cette carte est maintenant considérée comme obsolète, non en raison de ses capacités mais du fait qu'ACME Systems produit maintenant des modules bien plus puissants. Preuve de la validité toujours actuelle de la carte Fox Classic, voici un petit guide pour y installer OpenWrt.

La carte est construite autour d'un module MCM d'Axis Communications intégrant, entre autres choses, le processeur ETRAX 100LX CPU, 4 Mo de mémoire Flash, 16 Mo de SDRAM et une interface Ethernet avec son *transceiver*. Comme l'explique clairement la page du Wiki d'ACME Systems (http://foxlx.acmesystems.it/?id=30), il n'existe aucune différence fonctionnelle entre cette carte et la FOX Board LX, en dehors d'un ajustement de timing pour la SDRAM (`0x80608002` pour la Classic et `0x80008002` pour la LX). Visuellement, les cartes se différencient par la couleur du circuit imprimé et surtout par le fait que la LX comprend la mémoire Flash, la SDRAM et quelques autres éléments sous la forme de composants externes soudés sous la carte.

Il est important de bien comprendre ces différences, car sur la plupart des sites et pages web, les deux modèles de cartes sont peu ou pas différenciés, tantôt confondus et d'autres fois encore nommés de manière relativement aléatoire :

- ACME Classic : ACME Fox Classic en version MCM ;
- Foxboard 416 : FOX Board LX identique à la MCM (4 Mo Flash / 16 Mo SDRAM) ;
- Foxboard 816 et 832 : FOX Board LX avec 8 Mo de Flash et respectivement 16 Mo et 32 Mo de SDRAM. Du fait de l'abandon du module MCM, ACME Systems peut produire des cartes intégrant plus de Flash et de mémoire vive.

Les produits ACME utilisant les processeurs ou modules Axis reposent généralement sur un environnement dédié, open source et configurable à souhait. Cependant, si comme moi, vous avez pris l'habitude d'utiliser vos systèmes embarqués de façon plus ou moins modulaire via l'ajout/suppression de paquets, la solution d'ACME Systems n'est pas très satisfaisante. Fort heureusement, Claudio Mignanti s'est mis en tête, il y a quelque temps déjà, de porter une version spécifique d'OpenWrt sur la carte ACME, renommant au passage le projet en CrisOS. Le développement avançant bon train, ces modifications ont été finalement intégrées dans la branche de développement d'OpenWrt. Claudio est maintenant un *committer* OpenWrt et CrisOS est un projet officiellement arrêté en tant que tel.

1 CONFIGURATION ET CONSTRUCTION D'OPENWRT

Comme dit précédemment, le portage d'OpenWrt n'est disponible que dans la branche de développement du projet. On remarquera que le nombre de plates-formes supportées ne cesse de grandir. Il y a, en effet, presque deux fois plus d'entrées dans le `target/linux` de la version de développement que dans la dernière version stable (8.09.2). Parmi les plates-formes en question, on retrouve `etrax`. La première chose à faire est donc de récupérer une version SVN d'OpenWrt avec :

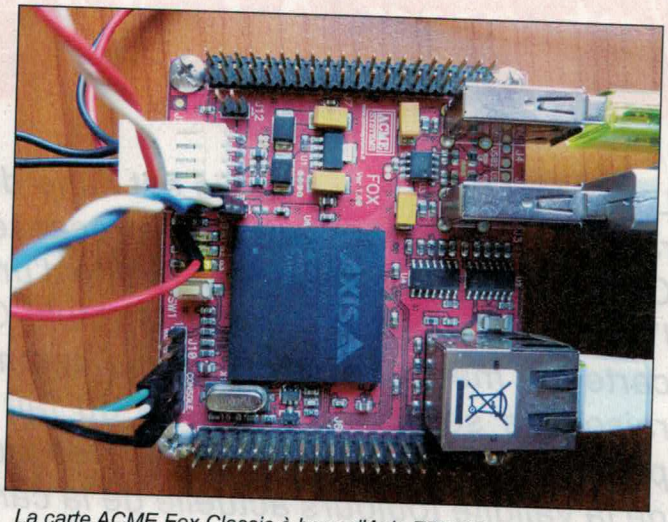
```
% svn co svn://svn.openwrt.org/openwrt/trunk/
```

On suppose ici que vous avez installé l'ensemble des outils de développement adéquats incluant, bien entendu, Subversion. Si ce n'est pas le cas, sur Debian GNU/Linux, il vous suffira d'installer les paquets `g++`, `libncurses5-dev`, `zlib1g-dev`, `bison`, `flex`, `unzip`, `autoconf`, `gawk`, `make`, `gettext`, `gcc`, `binutils`, `patch`, `bzip2`, `libz-dev`, `libc`, `headers`, `asciidoc` et `subversion`.

Il convient ensuite de mettre à jour les *feeds* OpenWrt avec :

```
% cd trunk
% ./scripts/feeds update
```

Le dépôt principal OpenWrt n'intègre que la base du système avec un jeu minimal d'outils et de bibliothèques. Le reste des éléments est maintenu en dehors du *truck* principal. Notez que la commande/cible `make package/symlinks`, souvent référencée sur le Web, est considérée aujourd'hui comme obsolète. Celle-ci est initialement prévue pour créer l'ensemble des liens symboliques entre les feeds et l'environnement de construction local. Si vous souhaitez chercher et ajouter un paquet se trouvant dans les feeds, vous devez désormais utiliser la commande `./scripts/feeds search` puis `./scripts/feeds install`. Exemple :



La carte ACME Fox Classic à base d'Axis ETRAX

```
% ./scripts/feeds search usb utils
Search results in feed 'packages':
ov51x-jpeg-utils   OV51x based USB webcam utilities
qc-usb-utils       Utility programs for the qc-usb kernel module
usbutils           USB devices listing utilities

% ./scripts/feeds install -d m usbutils
Installing package 'usbutils'
Installing package 'libusb'
```

L'option `-d` permet de définir l'état par défaut pour le paquet (`m` pour module et la fabrication d'un paquet `opkg`, `y` pour une intégration dans l'image qui sera construite, et `n` pour ne pas compiler du tout).

Vient ensuite la phase de configuration du système. Dans un premier temps, un petit `make defconfig` vous permettra d'obtenir une configuration par défaut et ainsi valider les dépendances de compilation. En cas de problème, lisez attentivement les messages à l'écran, repérez les dépendances non satisfaites et installez les paquets qui vous manquent sur votre distribution. Il est temps ensuite de passer aux choses sérieuses avec un `make menuconfig`. Vous vous retrouverez alors devant l'interface de configuration *curses* de Buildroot (voir article de P. Ficheux sur le sujet, dans le présent magazine). Là, choisissez ***Foxboard (ETRAX 100LX)*** dans le menu ***Target System***, puis parcourez l'ensemble de la configuration pour choisir les éléments qui vous sont nécessaires. Il est généralement tentant de laisser active la compilation sous la forme modulaire (`<M>`) de bon nombre d'éléments. Notez cependant que plus vous ferez le ménage dans les outils, fonctionnalités et bibliothèques, plus la construction du système sera rapide. Nous verrons plus loin qu'il est également possible de travailler avec plusieurs profils de configuration.

Avant de lancer la compilation/construction, il est nécessaire de faire un tour dans la configuration du noyau avec `make kernel_menuconfig`. En effet, la configuration par défaut est prévue pour une carte LX 416 et différents éléments ne correspondent pas. Dans l'interface qui se présente à vous, choisissez ***Hardware setup*** et assurez-vous des points suivants :

- ***Processor type*** doit être réglé sur ETRAX-100LX-v2 ;
- ***DRAM size*** sur `16` ;
- ***R_SDRAM_TIMING*** à `0x080608002` dans CRIS v10 options (pour la ACME Fox Classic).

Quittez ensuite l'interface, sans oublier d'enregistrer les changements, puis lancez la construction du système avec un simple `make` :

```
make[1] world
make[2] target/compile
make[3] -C target/linux compile
make[2] package/cleanup
make[2] package/compile
make[3] -C package/arptables compile
[...]
make[3] -C feeds/packages/libs/libusb compile
make[3] -C package/zlib compile
make[3] -C feeds/packages/utils/usbutils compile
[...]
make[3] -C target/linux install
make[6] -C target/linux/etrax/image/e100boot compile
make[6] -C target/linux/etrax/image/mkfimage compile
make[3] -C target/sdk install
make[3] -C target/imagebuilder install
make[3] -C target/toolchain install
make[2] package/index
```

Comme vous pouvez le voir, en fonction des options que vous aurez définies, les différents éléments seront compilés à commencer par la chaîne de compilation et les outils de construction du système (l'*Image Builder* et le SDK). On reconnaît ici deux lignes correspondant à la compilation de notre paquet explicitement installé via les feeds (`libusb` et `usbutils`). En fin de processus, on remarque également la compilation des outils qui vont nous permettre de préparer puis mettre à jour le système dans la flash de la carte (`mkfimage` et `e100boot`).

2 INSTALLATION DU SYSTÈME SUR LA CARTE ACME

Inutile de le préciser, OpenWrt est quelque chose de vraiment bien conçu et un projet totalement mature. En toute logique, vous trouverez les éléments intéressants pour votre carte dans le répertoire `bin/etrax`. Vous avez là :

- `boot_linux`, l'outil de flashage ;
- `openwrt-etrax-squashfs-fimage`, une image du système pour le mode SquashFS (système de fichiers compressé et en lecture seule) ;
- `openwrt-etrax-jffs2-64k-fimage`, une image JFFS2 en lecture/écriture.

Les deux images construites découlent du choix que vous aurez fait dans l'interface de configuration sous ***Target Images***. C'est également là que vous trouverez l'option ***Build fimage*** activée par défaut et vous permettant de créer une image directement utilisable avec la carte ACME. Vous remarquerez que ces images font très exactement 4 Mo, ce qui correspond à la taille de la mémoire Flash disponible sur la carte. L'installation à proprement parler se fera avec :

```
% cd bin/etrax
% sudo ./boot_linux -F -d br0 -i ./openwrt-etrax-jffs2-64k-fimage
Using internal boot loader: INTERNAL_NW - Network boot (default).
Starting boot...
We're doing a flash write, this may take up to a few minutes...

Device ID = 0xc0fa3de5
This bootloader was built by root on Wed May 16 21:31:41 CEST 2007.
Checksum of bootloader is 0x000a0a94
Waiting for load info.
Checksum of file is 0x00001e56
Got load info.
SET_REGISTER
0xb0000000
0x000095f8
[...]
0x803e0000: Erasing 0x00010000 bytes
0x803e0000: Writing 0x00010000 bytes
0x803f0000: Erasing 0x00010000 bytes
0x803f0000: Writing 0x00010000 bytes
0x80000000: Verifying...OK
JUMP
0x00000000
END
Exiting with code 0
```

A ce moment, vous devrez couper l'alimentation de votre carte FOX, placer un cavalier sur J8 et la remettre sous tension. Il est recommandé de connecter la carte directement avec une interface réseau de votre machine via un câble croisé (ou un câble droit si votre interface supporte l'auto-négociation). Comme vous pouvez le constater, ici l'interface précisée en argument de l'option **-d** est **br0**, ce qui correspond sans doute à la solution la plus risquée puisqu'il s'agit d'un bridge réunissant **eth0** et deux interfaces TUN/TAP (**eth0** étant bien entendu connecté à un switch sur lequel est branché la carte ACME). Malgré ceci, le flashage se passe sans le moindre problème et la carte redémarre automatiquement en mode « normal » (n'oubliez pas de retirer le cavalier ensuite, pour éviter de bloquer le prochain redémarrage).

Une console série connectée sur la carte ACME nous permet de prendre la main et de voir le démarrage du système avant d'arriver à l'invite du shell BusyBox :

```
 _______                     ________        __
|       |.-----.-----.-----.|  |  |  |.----.|  |_
|   -   ||  _  |  -__|     ||  |  |  ||   _||   _|
|_______||   __|_____|__|__||________||__|  |____|
         |__| W I R E L E S S   F R E E D O M
KAMIKAZE (bleeding edge, r19788) ------------------
  * 1Ø oz Vodka       Shake well with ice and strain
  * 1Ø oz Triple sec  mixture into 1Ø shot glasses.
  * 1Ø oz lime juice  Salute!
---------------------------------------------------
root@OpenWrt:/#
```

Ce premier démarrage sera sensiblement plus lent que les suivants. En effet, c'est là que la mémoire Flash non utilisée par le système de fichiers est réinitialisée comme le précisent les messages au démarrage :

```
jffs2_scan_eraseblock(): End of filesystem marker found at 0xf0000
jffs2_build_filesystem(): unlocking the mtd device... done.
jffs2_build_filesystem(): erasing all blocks after the end marker... done.
```

Selon les options de configuration définies par vos soins dans **kernel_menuconfig**, un certain nombre de périphériques intégrés sont pris en charge comme les trois ports séries :

```
ETRAX 100LX serial-driver 1.25 $, (c) 2000-2004 Axis Communications AB
ttyS0 at 0xb0000060 is a builtin UART with DMA
ttyS2 at 0xb0000070 is a builtin UART with DMA
ttyS3 at 0xb0000078 is a builtin UART with DMA
```

Les partitions MTD de la mémoire Flash :

```
Creating 2 MTD partitions on "cse0":
0x000000000000-0x0000000e0000 : "kernel"
0x0000000e0000-0x000000400000 : "rootfs"
mtd: partition "rootfs" set to be root filesystem
```

Ou encore le bus i2c directement géré par le SoC Axis :

```
I2C driver v2.2, (c) 1999-2004 Axis Communications AB
EEPROM char device v0.3, (c) 2000 Axis Communications AB
eeprom: Read failed to address the eeprom: 0x00000000 (0) page: 0
eeprom: Failed to read in 2k mode!
eeprom: Read failed to address the eeprom: 0x000001FB (507) page: 1
eeprom: Found i2c compatible 16kB eeprom.
```

3 CONFIGURATION DU SYSTÈME

A présent, puisque notre système a correctement démarré, il est temps de le configurer. Nous ne nous étendrons pas sur le sujet puisque la configuration d'OpenWrt a déjà fait l'objet de plusieurs articles portant principalement sur la plate-forme Atheros Fonera. Si, comme moi, dans la configuration par défaut d'OpenWrt, vous avez choisi de configurer le système via DHCP, votre fichier **/etc/config/network** doit ressembler à ceci :

```
config 'interface' 'lan'
        option 'ifname' 'eth0'
        option 'type' 'none'
        option 'dns' ''
        option 'proto' 'dhcp'
        option 'gateway' ''
        option 'netmask' ''
        option 'ipaddr' ''
```

Il n'est pas recommandé de modifier directement les fichiers dans **/etc/config**, mais plutôt de reposer sur l'outil de configuration **uci**. Un **uci show | grep lan** vous retournera les mêmes informations :

```
# uci show | grep lan
network.lan=interface
network.lan.ifname=eth0
network.lan.type=none
network.lan.dns=
network.lan.proto=dhcp
network.lan.gateway=
network.lan.netmask=
network.lan.ipaddr=
```

Vous pouvez dès lors ajouter une paire de clé/valeur afin de définir l'adresse MAC de l'interface réseau :

```
# uci set network.lan.macaddr="00:40:8C:CD:42:42"
# uci commit
```

Il ne vous reste plus qu'à configurer votre serveur DHCP avec l'adresse MAC correspondante, redémarrer la carte ACME et le tour est joué. Un message vous signale le changement d'adresse MAC :

```
eth0: changed MAC to 00:40:8c:cd:42:42
```

Et le serveur DHCP fait son travail en fonction de cette nouvelle adresse :

```
dhcpd: DHCPDISCOVER from 00:40:8c:cd:42:42 via br0
dhcpd: DHCPOFFER on 192.168.0.171 to 00:40:8c:cd:42:42 via br0
dhcpd: DHCPREQUEST for 192.168.0.171 (192.168.0.1) from
00:40:8c:cd:42:42 via br0
dhcpd: DHCPACK on 192.168.0.171 to 00:40:8c:cd:42:42 via br0
```

Notez que, par défaut, le serveur SSH Dropbear est lancé tout comme un serveur HTTP (NusyBox) et un **telnetd**. Ce dernier ne sera plus lancé dès lors que vous aurez défini un mot de passe pour l'utilisateur UID 0 (root). Vous pourrez alors vous connecter en SSH sous le compte root pour la suite des opérations et délaisser la console série relativement lente (même si elle est pas défaut en 115200 bps).

Vous disposez désormais d'une interface réseau fonctionnelle, vous pouvez donc vous pencher sur la personnalisation de la logithèque du système. Un système de gestion de paquets est disponible et préinstallé, c'est **opkg**. Vous pouvez d'ores et déjà lister les paquets en présence avec :

```
# opkg list_installed
base-files - 39-r19788
busybox - 1.15.3-1
dropbear - 0.52-4
hotplug2 - 1.0-beta-1
libc - 0.9.30.1-39
libgcc - 4.3.3-39
liblua - 5.1.4-5
libuci - 0.7.5-1
libuci-lua - 0.7.5-1
lua - 5.1.4-5
mtd - 11
opkg - 513-1
uci - 0.7.5-1
ucitrigger - 0.7.5-1
udevtrigger - 106-1
```

C'est le fichier **/etc/opkg.conf** qui vous permettra de définir le dépôt à utiliser et la manière de procéder. Le fichier installé par défaut pointe sur le site officiel, mais comme il s'agit d'une version de développement, mieux vaut reposer sur votre propre serveur pour les paquets. Pour cela, rien de plus simple puisqu'il nous suffit d'un serveur HTTP léger comme Lighttpd, nginx ou encore Cherokee (voir *GLMF* n°125 actuellement en kiosque). La configuration d'un simple alias proposant le contenu de **bin/etrax/packages** fera l'affaire et notre **opkg.conf** ressemblera alors à ceci :

```
src/gz packages http://192.168.0.1/etraxMCM
dest root /
dest ram /tmp
```

Un simple **opkg update** nous permettra de mettre à jour la liste des paquets disponibles et nous pourrons ensuite installer ceux qui nous manquent. Par exemple, ici, nous avons décidé de construire le paquet **usbutils** contenant entre autres choses la commande **lsusb**. Nous pouvons donc l'installer avec :

```
# opkg install usbutils
Installing usbutils (0.86-1) to root...
Downloading http://192.168.0.1/etraxMCM/usbutils_0.86-1_etrax.ipk.
Installing libusb (0.1.12-2) to root...
Downloading http://192.168.0.1/etraxMCM/libusb_0.1.12-2_etrax.ipk.
Installing zlib (1.2.3-5) to root...
Downloading http://192.168.0.1/etraxMCM/zlib_1.2.3-5_etrax.ipk.
Configuring usbutils.
Connecting to www.linux-usb.org (216.34.181.97:80)
-    100% |*****************************|   380k 00:00:00 ETA
Done.
Configuring zlib.
Configuring libusb.
```

Remarquez que notre espace de stockage est limité (1,5 Mo déjà consommés sur les 4 Mo disponibles) et que l'installation de certains paquets, comme **usbutils**, a tendance à occuper beaucoup de place (en raison de l'installation de la liste des VID/PID en double). Si le support de l'USB est bel et bien activé dans la configuration du noyau sous la forme de module ou autre, vos manipulations s'arrêtent là. Normalement, le pseudo-système de fichiers **/proc/bus/usb** est automatiquement monté. Si ce n'est pas le cas, vérifiez la prise en charge du contrôleur USB (les deux ports) dans les messages du noyau. Attention, certaines fonctionnalités de la carte ACME utilisent des lignes d'entrée/sortie communes avec d'autres fonctionnalités. En cas de conflit, la configuration du noyau ne vous préviendra pas. Ainsi, en activant la prise en charge de l'USB et du second port série (**ttyS1**), le démarrage du système échouera avec un magnifique *Oops* :

```
ETRAX 100LX serial-driver 1.25 $,
     (c) 2000-2004 Axis Communications AB^
ttyS0 at 0xb0000060 is a builtin UART with DMA
ttyS1 at 0xb0000068 is a builtin UART with DMA
ttyS2 at 0xb0000070 is a builtin UART with DMA
ttyS3 at 0xb0000078 is a builtin UART with DMA
IRQ 8/serial : IRQF_DISABLED is not guaranteed on shared IRQs
ETRAX 100LX 10/100MBit ethernet v2.0 (c) 1998-2007
     Axis Communications AB
eth0: changed MAC to 00:40:8c:cd:00:00
ETRAX 100LX USB Host Controller version 1.00-openwrt_diff
     (c) 2005, 2006 Axis Communications AB
usb_devdrv: Etrax 100LX USB Revision 16 v1,2
usb_devdrv: Bulk timer interval, start:2 eot:6
cris_request_io_interface: group E needed by usb_1 not available
usb-host: request IO interface usb1 failed
hc-crisv10 hc-crisv10.0: reset
hc-crisv10 hc-crisv10.0: remove, state 0
Unable to handle kernel NULL pointer dereference
Oops: 0000
```

Les ports USB1 et ttyS1 partagent, en effet, les mêmes lignes sur les mêmes ports. Pour utiliser l'un, il vous faudra désactiver l'autre et inversement.

En parlant de configuration et d'USB, vous conviendrez avec moi que 4 Mo d'espace de stockage est quelque chose de relativement réduit, même pour un système embarqué. En cas de besoin, vous pouvez cependant tirer profit des deux ports USB en y plaçant des clés USB. Une fois initialisée avec un système de fichiers adéquat, vous pourrez les utiliser comme espace de stockage sur des points de montage comme **/var**, **/opt** ou **/home**. Là encore, c'est **uci** qu'il faudra utiliser ou, au pire, modifier **/etc/config/fstab** mais surtout pas **/etc/fstab** comme sur un système classique. Ajouter un profil de montage dans la configuration **uci** est relativement simple :

```
# uci add fstab mount
cfg064d78
# uci show fstab
[...]
fstab.@mount[1]=mount

# uci add_list fstab.@mount[1].device=/home
# uci add_list fstab.@mount[1].device=/dev/sdb1
# uci add_list fstab.@mount[1].fstype=ext3
# uci add_list fstab.@mount[1].options='defaults,noatime'
# uci add_list fstab.@mount[1].enabled=1

# uci show fstab
[...]
fstab.@mount[1]=mount
fstab.@mount[1].target=/home
fstab.@mount[1].device=/dev/sdb1
fstab.@mount[1].fstype=ext3
fstab.@mount[1].options='defaults,noatime'
fstab.@mount[1].enabled=1
```

Notez l'utilisation de l'option **noatime**, permettant de grandement réduire le nombre d'opérations d'écriture. Ceci aura pour effet de prolonger sensiblement la durée de vie de la mémoire Flash d'une clé USB.

4 FACILITÉ DE GESTION DES CONSTRUCTIONS

OpenWrt, depuis la première version de Kamikaze, permet de prendre en charge la gestion des environnements de construction. Comprenez par là qu'il est possible, avec un seul environnement, d'utiliser plusieurs profils de configuration et de basculer de l'un à l'autre sans avoir à reconfigurer l'ensemble.

Si, comme moi, vous procédez par étape dans la construction de votre système, il est fort probable que vous soyez tenté de multiplier les copies du répertoire de travail. Ainsi, vous disposez de plusieurs versions du système cible. Fort heureusement, il est possible de procéder de manière plus élégante avec le script **./scripts/env**.

Une fois votre première configuration terminée, satisfaisante, testée et fonctionnelle, utilisez la commande **./scripts/env new premiere**. Ceci aura pour effet de copier les fichiers **.config** et **.files** dans le sous-répertoire **env** et de créer les liens symboliques correspondants. Dès lors, vous pourrez changer votre configuration à souhait tout en gardant une marge de sécurité. A tous moments, vous pourrez :

- comparer la version en cours avec votre copie enregistrée : **./scripts/env diff** ;
- enregistrer les changements effectués : **./scripts/env save** ;
- revenir à la version précédente : **./scripts/env revert** ;
- créer une nouvelle version : **./scripts/env new autre_prof**.

Cette dernière commande vous demandera alors si vous préférez créer une copie de l'environnement sélectionné ou créer un nouveau profil en utilisant la configuration actuelle. Il sera ensuite possible de passer d'un profil à un autre avec **./scripts/env switch nom_prof**.

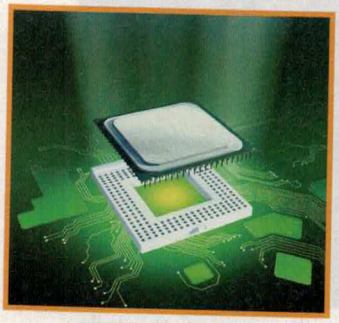
Il devient alors extrêmement simple de travailler avec plusieurs versions différentes de votre système et, par la même occasion, de gérer correctement le temps de construction dont vous disposez. Attention, la configuration spécifique au noyau ne sera pas prise en charge par cette gestion. Il s'agit uniquement de la configuration propre à OpenWrt/Buildroot.

CONCLUSION

Comme vous pouvez le voir, la disponibilité d'une version d'OpenWrt prenant en charge les « anciennes » cartes ACME Systems offre une nouvelle jeunesse à cette gamme de produits. A l'heure où nous écrivons ces lignes, la plus puissante des cartes à base d'ETRAX, la FOX Board LX 8+32, est annoncée comme disponible jusqu'à mars 2010 sur le eShop d'ACME Systems. En effet, il n'y a aucune différence de coût entre celle-ci et la nouvelle FOX Board G20 (ARM9/AT91SAM9G20, 8 Mo de Flash et 64 Mo de SDRAM) à laquelle il faudra cependant ajouter une carte micro SD. L'objectif principal de l'utilisation d'OpenWrt sur cette carte est de disposer d'un système, certes vieillissant, mais parfaitement fonctionnel et modulaire. Bien entendu, pour de nouveaux projets, il vaudra mieux se tourner vers la G20. Pour ma part, ma « vieille » Foxboard servira de base domotique pour une partie de mon chez moi et pilotera différents montages et capteurs qui deviendront ainsi accessibles en Wifi (clé RT73). Ceci fera peut-être l'objet d'un article spécifique le moment venu.

Auteur : Denis Bodor

Mise en œuvre de Linux embarqué sur



Sans nous en rendre compte, nous utilisons actuellement tous de nombreux systèmes embarqués. En effet, de nos jours, ces systèmes électroniques sont présents dans de nombreux domaines d'application tels que les systèmes multimédias, la domotique, la voiture, l'avionique...

Auteurs

- Ahmed Ben Atitallah avec la participation de Patrice Kadionik

Le Libre est actuellement en train de prendre de plus en plus de place dans le développement des systèmes embarqués. Les initiatives open source sont nombreuses dans le domaine des systèmes embarqués, des processeurs softcore sous forme de blocs IP (*Intellectual Property*), des chaînes de compilation croisées et de Linux embarqué.

Cet article décrit la mise en œuvre d'une plate-forme de développement à base du processeur softcore libre LEON et de Linux embarqué comme un système d'exploitation, afin de gérer les tâches du système embarqué réalisé.

1 L'EMBARQUÉ ET L'OPEN SOURCE

Un système embarqué peut être défini comme « un système électronique et informatique autonome ne possédant pas des Entrées/Sorties standards comme un clavier ou un écran d'ordinateur».

1.1 Les contraintes des systèmes embarqués

Les contraintes liées à l'électronique embarquée sont :

- Les contraintes physiques : le système embarqué doit présenter un faible encombrement et un faible poids.
- La dissipation de puissance : la dissipation est directement reliée à l'autonomie du système embarqué.
- Le temps de développement : en termes de développement de l'architecture système et des fonctions logicielles associées.
- Le coût : le système ne doit pas être cher, tout en tenant compte des performances à atteindre.

Dans le développement des systèmes embarqués, des choix doivent être réalisés pour satisfaire un ou plusieurs de ces critères. Par exemple, pour un ordinateur portable, l'autonomie est souvent négligée par rapport aux contraintes de performances et de taille. Par contre, pour un téléphone portable, on a intérêt à accroître l'autonomie.

1.2 Les SoC et l'embarqué

La conception des systèmes numériques embarqués est basée sur des SoC (*System on Chip*) ou SoPC (*System on Programmable Chip*).

Le développement des SoC et des SoPC se base sur l'usage d'un langage de description matérielle textuel tel que VHDL (*Very high speed integrated circuit Hardware Description Language*) ou Verilog, avec la mise en œuvre de composants matériels déjà élaborés. Les blocs IP (*Intellectual Property*) sont généralement catalogués et échangés par Internet en respectant les standards VSIA (*Virtual Socket Interface Alliance*) et sont disponibles en libre téléchargement, par exemple à partir du site www.opencore.org. Par contre, les modules IP commerciaux sont référencés sur le site www.us.design-reuse.com.

On peut définir un SoC ou un SoPC comme un ensemble de blocs IP intégrés dans un composant électronique avec au moins un processeur comme élément de traitement.

La figure 1 illustre un exemple de SoC. Dans ces systèmes miniatures, on trouve :

- des cœurs de processeurs (LEON, NIOS II, MicroBlaze, PowerPC, ...) ;
- des bus de communication ;
- des mémoires ;
- des contrôleurs vidéo, audio, ...

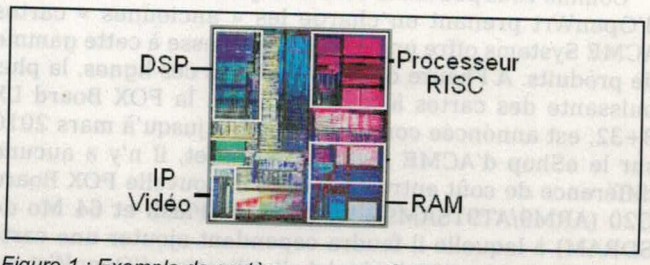


Figure 1 : Exemple de système sur puce SoC

le processeur softcore libre LEON

Dans le contexte de l'embarqué, on privilégie les processeurs qui sont écrits dans un langage de description matérielle (VHDL/Verilog), dont le code source peut être librement distribué et implanté dans n'importe quel circuit programmable FPGA (*Field Programmable Gate Array*), c'est-à-dire des processeurs sous forme d'un bloc IP libre.

Pour les processeurs softcore libres, on trouve principalement les processeurs LEON, OpenRisc, F-CPU, LatticeMico32, ...

Ce sont généralement des processeurs 32 bits ayant une architecture de type Harvard, avec un jeu d'instructions réduit RISC.

1.3 L'OS pour l'embarqué

Vue la complexité croissante des systèmes de traitement de l'information, on est arrivé à un stade où il devient impossible de gérer les ressources matérielles d'une architecture embarquée sans avoir recours à une logique programmée (logiciel) plus ou moins complexe. Cette interface logicielle est appelée système d'exploitation ou OS (*Operating System*).

Un système d'exploitation offre ainsi différents services pour mieux appréhender la complexité des systèmes embarqués :

- Apport du multitâche. Une application monolithique est divisée en une somme de tâches coopératives (système multitâche) ;
- Maîtrise des contraintes temporelles (système Temps Réel) ;
- Masquage des spécificités du matériel. On y accède de façon homogène et standard ;
- Développement de pilotes de périphérique (driver) simplifié pour pouvoir avoir accès aux accélérateurs matériels ;
- Apport d'un système de fichiers ;
- Possibilité de communications réseau : pour un contrôle du système à distance, par exemple.

Linux comme système d'exploitation devient de plus en plus prépondérant dans l'univers des systèmes embarqués.

On retrouve Linux dans l'embarqué pour les raisons suivantes :

- Logiciel libre disponible gratuitement au niveau source ;
- On a un système d'exploitation multitâche, un système de fichiers à disposition et une connectivité TCP/IP en standard ;
- Fiabilité et stabilité reconnues du système ;
- Il est possible d'avoir des versions Temps Réel ;
- Support d'un grand nombre de processeurs autres que x86 : Power PC, ARM, MIPS, LEON, NIOS II, MicroBlaze, ... ;
- Taille du noyau modeste et compatible avec les tailles de mémoires utilisées dans un système embarqué ;
- Différentes distributions proposées suivant l'usage : routeur IP, PDA, téléphone, ...

Linux pour l'embarqué existe pour 2 catégories de processeurs (32 bits minimum) :

- Processeur avec MMU (*Memory Management Unit*) : Linux embarqué, version Linux standard ;
- Processeur sans MMU : µClinux, version Linux adaptée.

Par la suite, on s'intéresse à la mise en place d'un environnement embarqué à base de modules IP libres. La plate-forme matérielle s'articule autour d'une carte STRATIX II d'Altera (on peut aussi utiliser des cartes de Xilinx). Le cœur du système met en œuvre le module IP softcore LEON et Linux comme système d'exploitation, pour gérer les différents périphériques et les tâches du système embarqué ainsi réalisé.

2 LE PROCESSEUR SOFTCORE LEON

2.1 Présentation du processeur LEON

LEON est un processeur RISC 32 bits. Il a été développé par l'ingénieur suédois Jiri GAISLER pour l'ESA (Agence Spatiale Européenne). Il a été conçu initialement comme un processeur compatible avec l'architecture SPARC V8 (*Scalable Processor ARChitecture*) avec une tolérance aux erreurs pour les applications spatiales (électronique durcie).

Il est actuellement disponible en deux versions destinées aux applications embarquées :

- Une version libre sous licence LGPL pour LEON 2 et GPL pour LEON 3, disponibles pour l'éducation ou la recherche et téléchargeables à partir du site http://www.gaisler.com.
- La version commerciale nécessite de payer une licence.

2.2 Caractéristiques du softcore LEON 3

LEON 3 est un processeur softcore qui est écrit en langage VHDL et qui est fourni comme composant de la bibliothèque GRLIB contenant plusieurs blocs IP matériels (contrôleur ethernet, unité de calcul en virgule flottante, contrôleur DMA, ...) et permettant de construire un système sur puce SoC ou SoPC.

LEON 3 est synthétisable sur des circuits numériques de type FPGA, comme ceux de Xilinx ou Altera, ou sous forme d'un circuit spécifique ASIC. Il fonctionne jusqu'à 125 MHz sur FPGA et 400 MHz sur un ASIC 0.13 µm.

Le softcore LEON 3 a les caractéristiques suivantes :

- Instructions aux normes SPARC V8 ;
- 7 étages de pipeline ;
- Cache d'instructions et cache de données (Architecture Harvard) ;
- Cache configurable : 1 à 4 étages, 1 à 256 Ko par étage ;
- Multiplication et division câblées ou non ;
- Interface de bus AMBA 2.0 ;
- MMU ;
- FPU ;
- Utilisable en SMP (plusieurs processeurs).

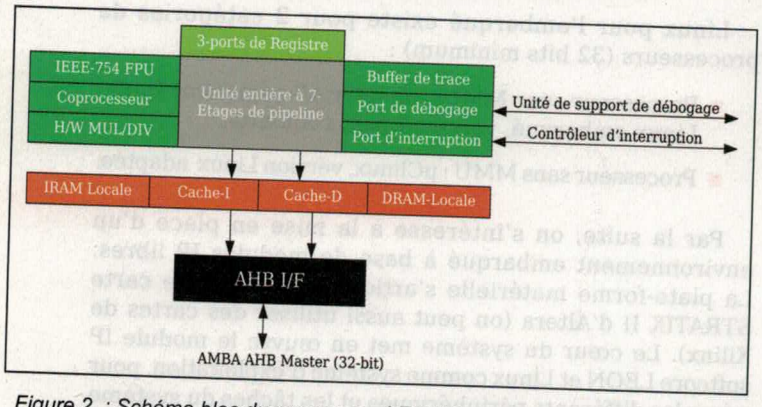


Figure 2. : Schéma bloc du processeur LEON 3

La figure 2 illustre le schéma bloc du processeur LEON 3 avec toutes ses fonctionnalités. Certaines de ces fonctionnalités peuvent être enlevées soit parce que l'application visée ne les nécessite pas, soit pour générer des architectures plus petites en surface de silicium ou pour pouvoir atteindre des fréquences de fonctionnement plus élevées.

Du point de vue logiciel, on a à disposition les outils suivants :

- On peut choisir comme plate-forme de développement Quartus II d'Altera ;
- Simulateur ModelSim de Mentor Graphics ;
- Chaîne d'outils GNU (compilation croisée) ;
- Systèmes d'exploitation supportés : SnapGear Linux, eCos, RTEMS.

2.3 Configuration et mise en place du processeur LEON 3

La mise en place du processeur LEON 3 sur une plate-forme FPGA passe par trois étapes.

La première étape consiste à configurer le processeur LEON 3, ce qui revient à configurer la bibliothèque GRLIB, qui est disponible en libre téléchargement à partir du site http://www.gaisler.com/cms/index.php?option=com_content&task=view&id=156&Itemid=104

La configuration du processeur est effectuée sous Linux ou sous Windows en utilisant l'environnement Cygwin (http://www.cygwin.com) à travers la commande `make xgrlib` en mode console (figure 3), afin de choisir les paramètres de LEON 3 tels que la taille des caches (instructions et données), MUL/DIV câblées ou non, utilisation de MMU, FPU, ..., ainsi que les blocs IP matériels pour la gestion des différents périphériques du système.

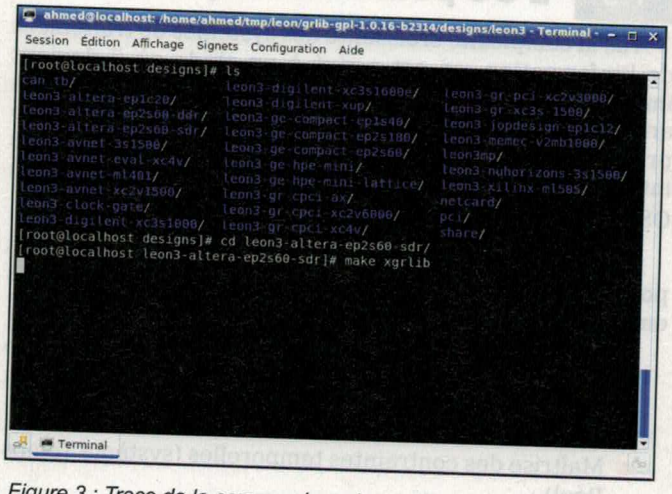


Figure 3 : Trace de la commande make xgrlib

La commande `make xgrlib` permet donc de lancer la fenêtre suivante :

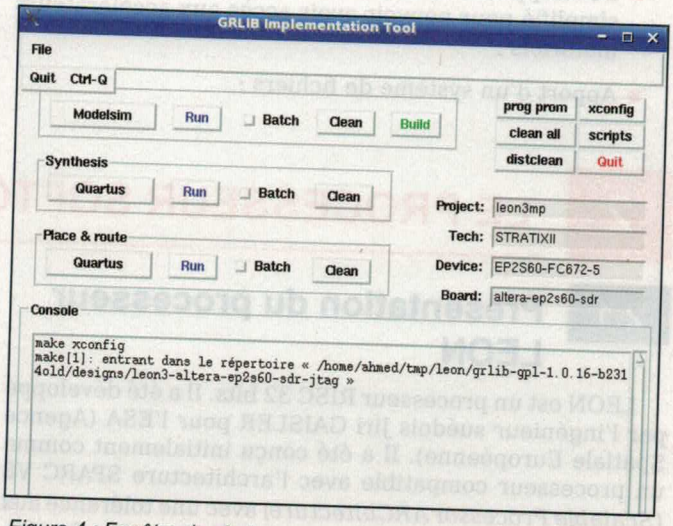


Figure 4 : Fenêtre de choix

A travers cette fenêtre, on peut choisir l'outil de simulation (ModelSim) et l'outil de synthèse et de placement routage (Quartus II) pour la conception de notre système. Ainsi, on peut configurer le processeur LEON 3 et ses périphériques en cliquant sur le bouton « xconfig » ou en mode console à travers la commande `make xconfig` (figures 5, 6 et 7).

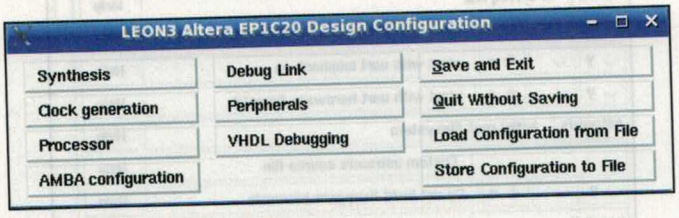


Figure 5 : Fenêtre de configuration

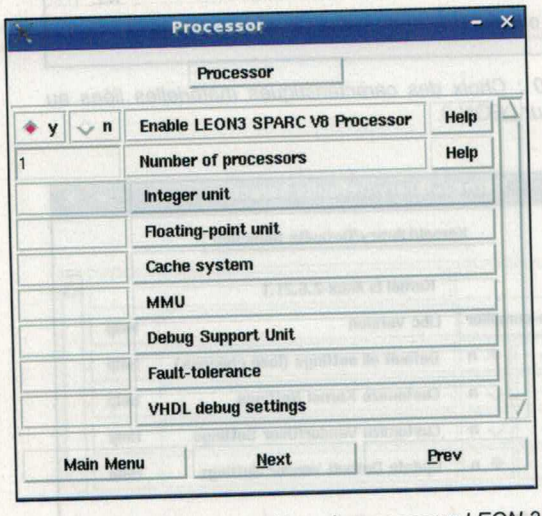


Figure 6 : Choix des paramètres du processeur LEON 3

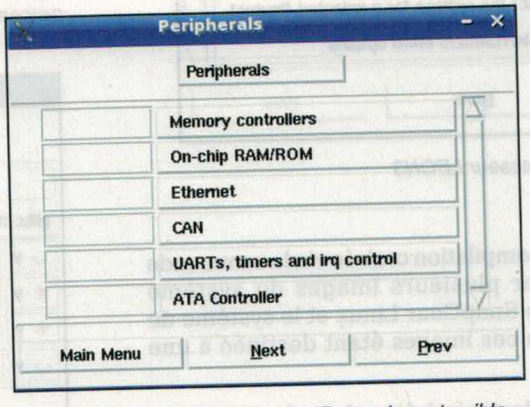


Figure 7 : Choix des modules IP pour la carte cible

La deuxième étape consiste à adapter les Entrées/Sorties du processeur LEON 3 aux périphériques disponibles sur la carte : SDRAM, SRAM, ports série, port ethernet, afficheur 7 segments, LED, boutons poussoirs, ...

Dans ce cas, si la carte est supportée par la bibliothèque GRLIB, il suffit de sélectionner la carte cible pendant la configuration du système (*Board Support Package* disponible). Si ce n'est pas le cas, on doit se référer aux documents techniques et à la schématique de la carte.

La troisième étape consiste à utiliser Quartus II (version Webpack), qu'on peut télécharger gratuitement à partir du site https://www.altera.com/support/software/download/altera_design/quartus_we/dnl-quartus_we.jsp afin de générer le projet en intégrant tous les modules IP. Après synthèse et routage, on a alors le fichier .sof de programmation du circuit FPGA correspondant au design SoPC. Il est possible aussi de simuler le design en utilisant ModelSim (version d'évaluation), qui est disponible à partir du site http://www.model.com/downloads/evaluations.asp.

Il reste maintenant à générer la partie logicielle afin de piloter les périphériques d'Entrées/Sorties du système en utilisant Linux, et plus précisément la distribution Linux SnapGear, qui est disponible sous licence GPL pour le processeur LEON 3.

3 MISE EN ŒUVRE DE LINUX SUR LE PROCESSEUR LEON 3

SnapGear Linux est une distribution Linux développée par la société CYBERGARD sous licence GPL pour les processeurs embarqués avec ou sans MMU, afin d'offrir au développeur plus de souplesse dans le choix de l'architecture cible.

En fait, SnapGear Linux offre deux noyaux : le noyau µClinux pour les processeurs dépourvus de MMU et le noyau Linux pour les processeurs avec MMU.

Pour la mise en place de SnapGear Linux sur LEON 3, il convient de définir le design de référence, qui servira après synthèse à programmer le composant FPGA de la carte cible, ici la carte Altera Stratix 2S60.

On a alors, après programmation du circuit FPGA Stratix II, connecté à l'ensemble des périphériques de la carte cible, la configuration matérielle suivante :

- CPU LEON3 32 bits avec MMU et FPU ;
- Interface SRAM (1 Mo) ;
- Interface FLASH (16 Mo) ;
- Interface SDRAM (16 Mo) ;
- Timer ;
- UART pour avoir un port série de communication ;
- Interface CompactFlash (64 Mo).

La mise en place d'une plate-forme de développement croisée nécessite le téléchargement de la version de SnapGear Linux portée pour LEON 3, le compilateur croisé `sparc-linux-gcc` ainsi que l'outil de débogage GRMON du site de Gaisler Research disponible à l'adresse

http://www.gaisler.com/cms/index.php?option=com_content&task=view&id=115&Itemid=103

La configuration de SnapGear Linux peut s'effectuer à travers la commande `make xconfig` sur le PC hôte sous Linux. Une série de fenêtres apparaît pour configurer le noyau SnapGear Linux et choisir ses applications (figures 8, 9, 10 et 11).

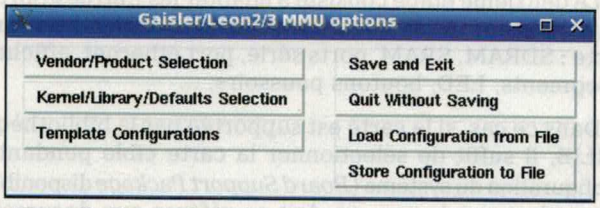


Figure 8 : Menu principal

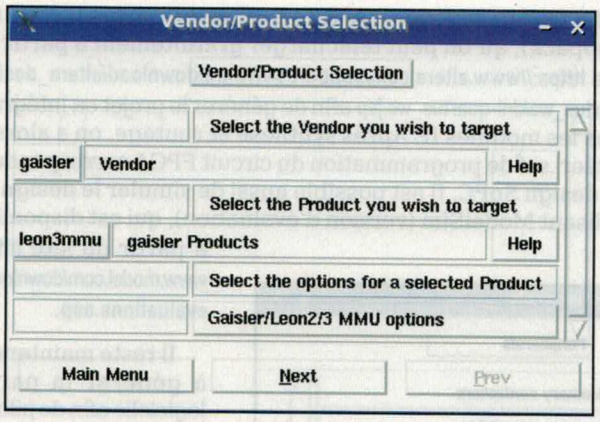


Figure 9 : Choix du processeur LEON3

Ensuite, à l'issue de la compilation croisée via la commande `make`, on peut récupérer plusieurs images du système Linux contenant le noyau SnapGear Linux et le système de fichiers root, chacune de ces images étant destinée à une utilisation particulière :

- `image.dsu` : image destinée à être chargée via le DSU (*Debug Support Unit*) ;
- `image.tsim` : image destinée à être simulée ;
- `image.dis` : c'est une version désassemblée du système ;
- `image.flashbz` : image destinée à être flashée.

Enfin, la dernière étape consiste à charger le fichier image `image.dsu` via le port *Debug Support Unit*. Cette étape suppose que le circuit FPGA est déjà programmé via le programmateur de Quartus II (figure 12) avec le fichier `.sof` contenant l'architecture matérielle compatible avec la partie logicielle.

Durant cette étape, on doit lancer l'outil de débogage nommé GRMON afin de pouvoir communiquer avec la carte cible en mode debug à travers le JTAG ou le port série (figure 13). A partir de cette interface, on effectue le chargement de l'image du système `image.dsu` dans la mémoire SDRAM, puis on exécute Linux via la commande `run`.

L'accès à Linux se fera à travers l'outil Minicom sous Linux ou Hyper Terminal sous Windows. Ces outils nous permettent de communiquer avec la carte cible via le port série (38400, 8, N, 1). La figure 14 montre le fonctionnement de SnapGear Linux sur le processeur LEON 3.

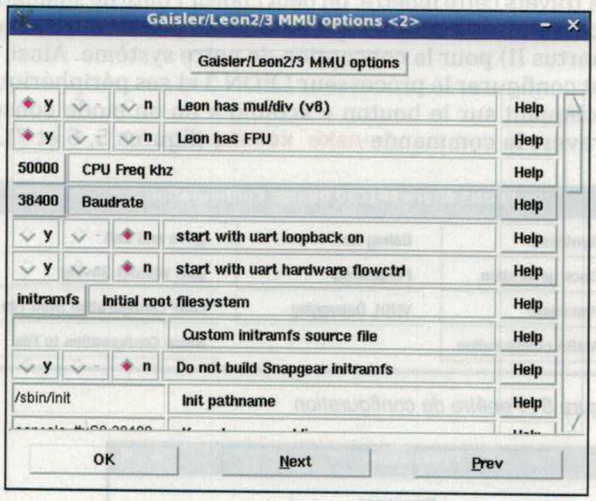


Figure 10 : Choix des caractéristiques matérielles liées au processeur LEON 3

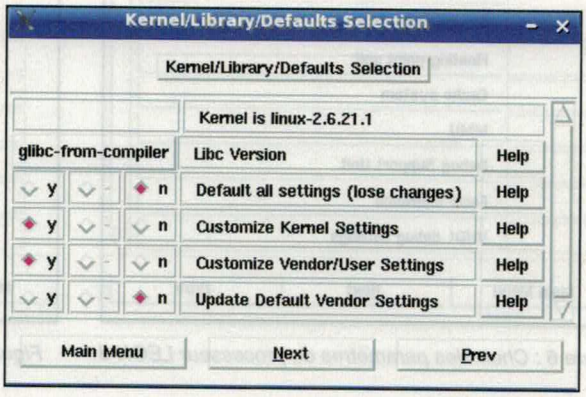


Figure 11 : Configuration du noyau et de ses applications

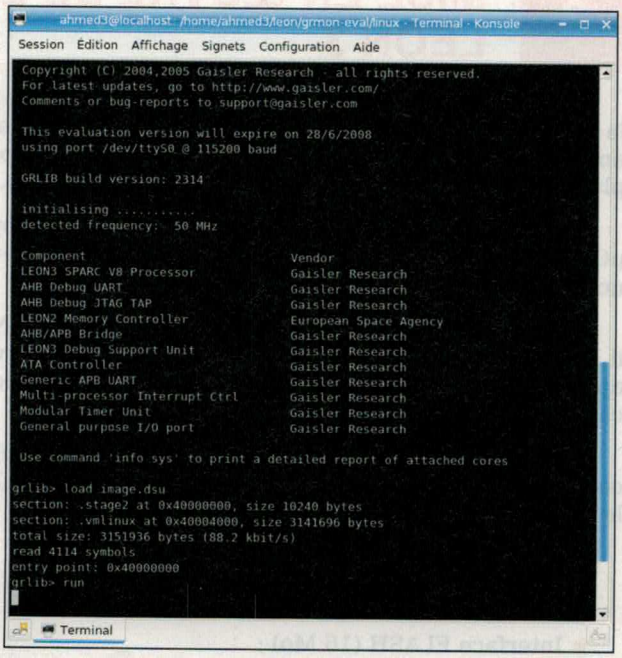


Figure 13 : Chargement puis exécution de Linux à travers le port de débogage

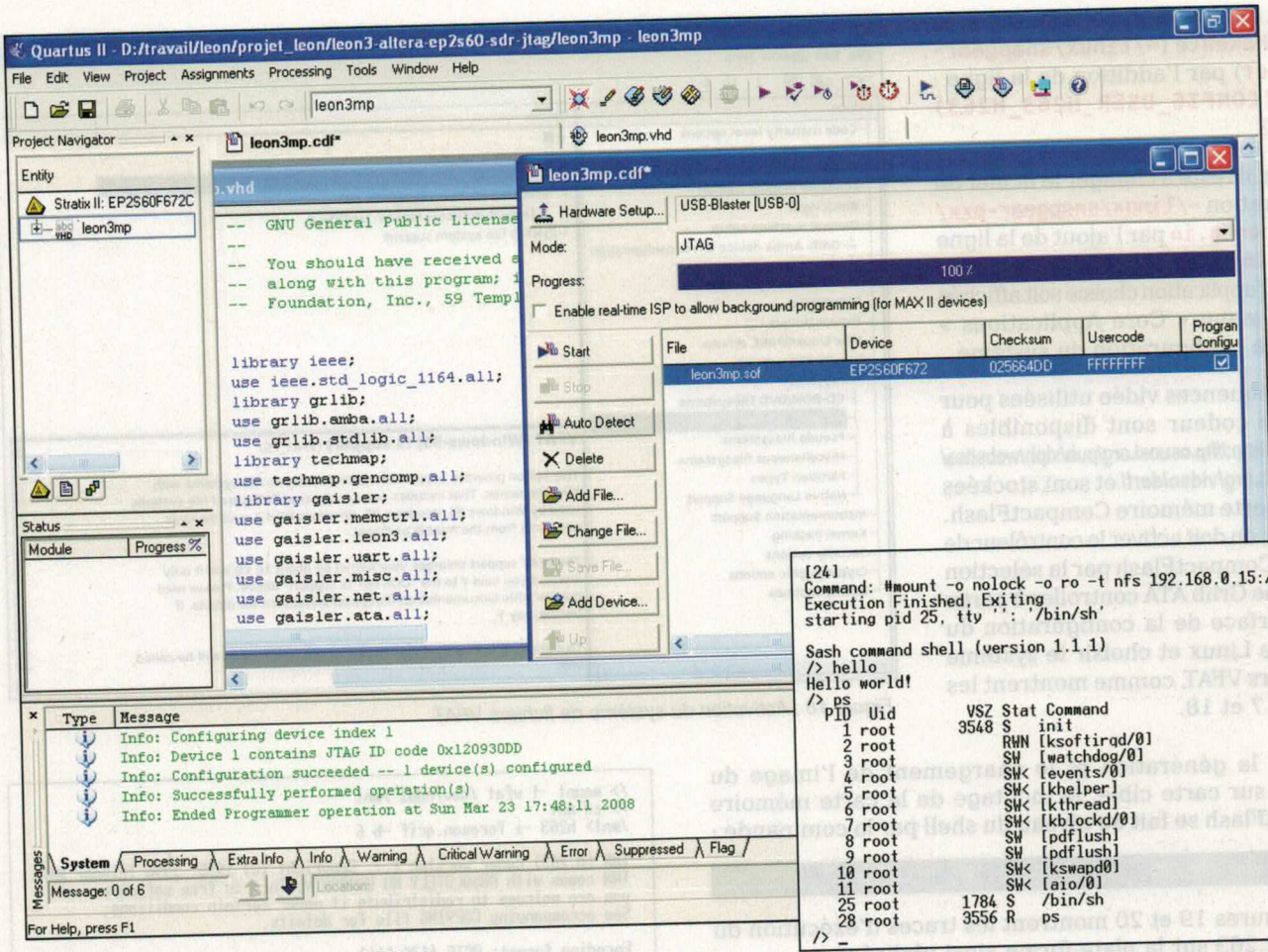


Figure 12 : Programmation du circuit FPGA

```
[24]
Command: #mount -o nolock -o ro -t nfs 192.168.0.15:/home /home/nfs
Execution Finished, Exiting
starting pid 25, tty '': '/bin/sh'

Sash command shell (version 1.1.1)
/> hello
Hello world!
/> ps
  PID  Uid        VSZ Stat Command
    1 root      3548 S    init
    2 root           RWN  [ksoftirqd/0]
    3 root           SW   [watchdog/0]
    4 root           SW<  [events/0]
    5 root           SW<  [khelper]
    6 root           SW<  [kthread]
    7 root           SW<  [kblockd/0]
    8 root           SW   [pdflush]
    9 root           SW   [pdflush]
   10 root           SW<  [kswapd0]
   11 root           SW<  [aio/0]
   25 root      1784 S    /bin/sh
   28 root      3556 R    ps
/> _
```

Figure 14 : SnapGear Linux sur LEON 3 en action !

4 EXEMPLE D'APPLICATION SNAPGEAR LINUX : LE CODEUR VIDEO H.263

Un exemple d'application sous SnapGear Linux - LEON3 est la mise en place d'un codeur de traitement vidéo tel que le codeur H.263 disponible en libre téléchargement à l'adresse

http://euler.slu.edu/~fritts/mediabench/mb2/mediabench2_video/h263enc/tmn-1.7.tar.gz

L'application a été ajoutée au menu « Core Applications » (figure 15) afin qu'on puisse la sélectionner lors de la configuration dans le choix des paquetages à utiliser.

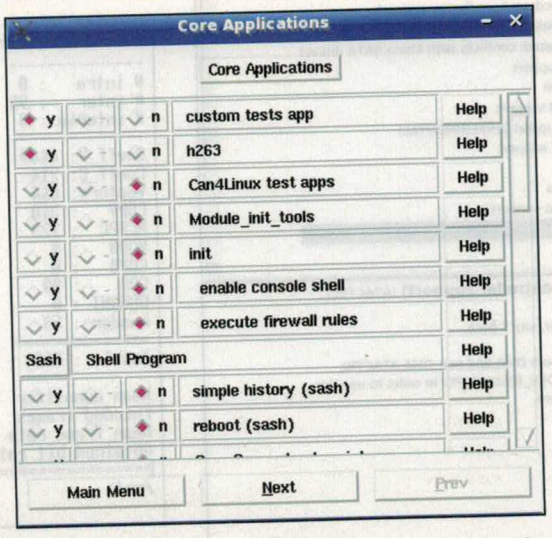


Figure 15 : Menu « Core Applications » contenant le codeur H.263

Pour définir l'application vidéo dans le menu « Core Applications », on va associer un fichier **Makefile** au codeur H.263, comme le montre la figure 16.

```
EXEC = h263
OBJS = main.o io.o dct.o coder.o quant.o mot_est.o pred.o  snr.o \
       countbit.o putbits.o ratectrl.o sac.o
       putvlc.o
all: $(EXEC)
$(EXEC): $(OBJS)
  $(CC) $(LDFLAGS) -o $@ $(OBJS) $(LDLIBS$(LDLIBS_$@)) -lm
romfs:
  $(ROMFSINST) /bin/$(EXEC)
clean:
  -rm -f $(EXEC) *.elf *.gdb *.o
```

Figure 16 : Makefile pour le codeur H.263

Les fichiers du codeur avec le fichier **Makefile** seront placés dans l'arborescence suivante : **~/linux/snapgear-pxx/user/h263**

Puis, on ajoute l'application vidéo au fichier **Makefile** (**~/linux/snapgear-pxx/user**) par l'addition de la ligne : **dir_$(CONFIG_USER_H263_H263) += h263**

Enfin, il reste à changer le fichier de configuration **~/linux/snapgear-pxx/config/config.in** par l'ajout de la ligne **bool 'h263' CONFIG_USER_H263_H263** afin que l'application choisie soit affichée dans le menu « Core Applications » lors de la configuration du système.

Les séquences vidéo utilisées pour tester le codeur sont disponibles à l'adresse http://ftp.osuosl.org/pub/xiph/websites/media.xiph.org/video/derf/ et sont stockées dans la carte mémoire CompactFlash. Pour ceci, on doit activer le contrôleur de la carte CompactFlash par la sélection de la ligne Grlib ATA controller à partir de l'interface de la configuration du noyau de Linux et choisir le système de fichiers VFAT, comme montrent les figures 17 et 18.

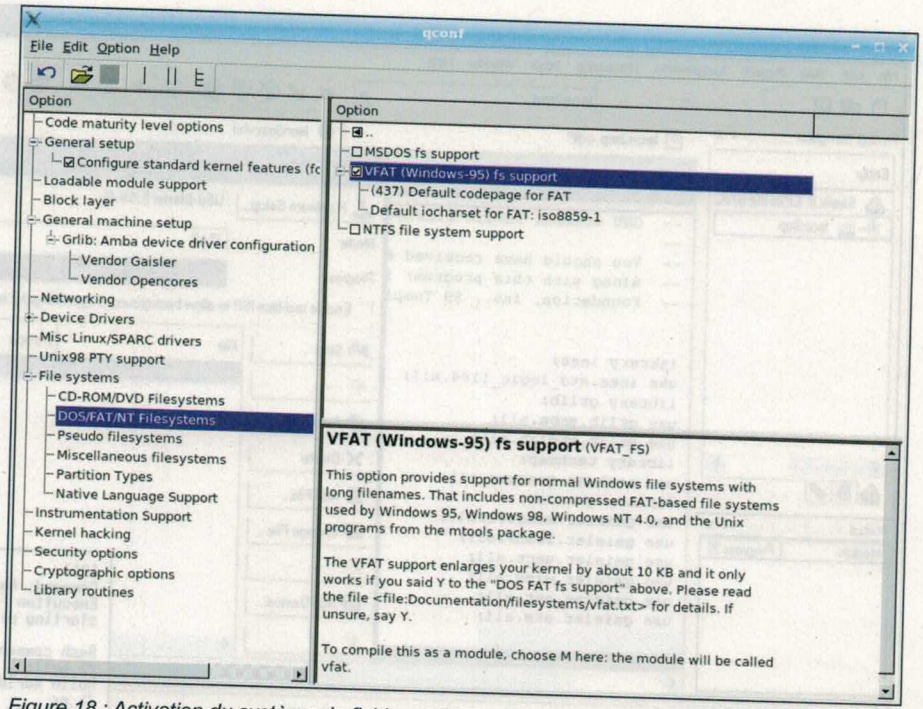


Figure 18 : Activation du système de fichiers VFAT

Après la génération et le chargement de l'image du système sur carte cible, le montage de la carte mémoire CompactFlash se fait au niveau du shell par la commande :

```
# mount -t vfat /dev/hda1 /mnt.
```

Les figures 19 et 20 montrent les traces d'exécution du codeur H.263 sur la plate-forme ainsi réalisée.

```
/> mount -t vfat /dev/hda1 /mnt
/> cd mnt
/mnt> h263 -i foreman.qcif -b 6

TMN (H.263) coder version 1.7, Copyright (C) 1995, 1996 Telenor R&D.
TMN comes with ABSOLUTELY NO WARRANTY; This is free software, and
you are welcome to redistribute it under certain conditions;
See accompanying COPYING file for details.

Encoding format: QCIF (176x144)
Encoding frame rate  : 10.00
Reference frame rate : 30.00
Orig. seq. frame rate: 30.00

Reading image no: 0
Coding...
```

Figure 19 : Exécution du codeur H.263 sur la séquence Foreman

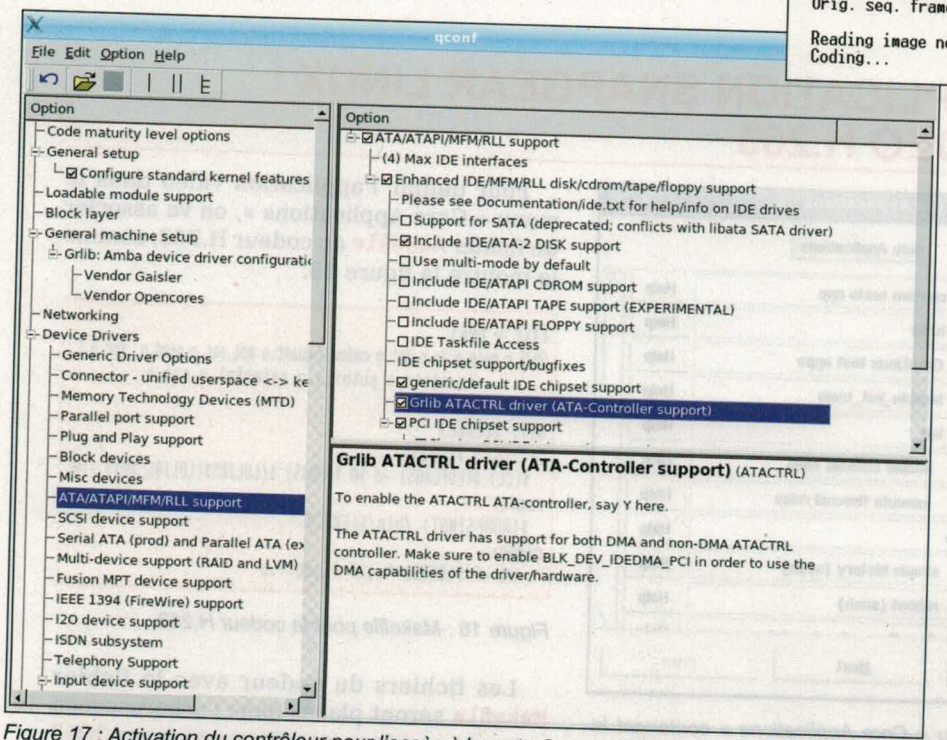


Figure 17 : Activation du contrôleur pour l'accès à la carte CompactFlash

```
-------------
# intra    : 0
# inter    : 95
# inter4v  : 0
-------------
Coeff_Y: 4449
Coeff_C: 214
Vectors: 844
CBPY   : 340
MCBPC  : 123
MODB   : 0
CBPB   : 0
COD    : 99
DQUANT : 0
header : 53
=============
Total  : 6124

Mean quantizer   : 10.00
Encoded frames   :   3 (  2)
Mean frame rate  : 10.00 Hz
Obtained bit rate: 117.89 (61.24) kbit/sec
=========================================
/mnt> _
```

Figure 20 : Résultats d'exécution du codeur H.263

CONCLUSION

Il est possible maintenant de concevoir un système embarqué permettant de gérer différents types d'applications embarquées en utilisant du matériel libre couplé à du logiciel libre.

Le choix d'un système d'exploitation comme Linux nous permet d'avoir un système embarqué incluant les fonctionnalités de base du noyau Linux, mais avec la possibilité d'étendre les fonctionnalités de ce système en utilisant des briques logicielles issues du monde du Libre.

La conception des systèmes embarqués nécessite maintenant la mise en œuvre d'accélérateurs matériels sous forme de blocs IP qu'on peut développer en VHDL ou Verilog, ou récupérer sous forme de blocs IP libres.

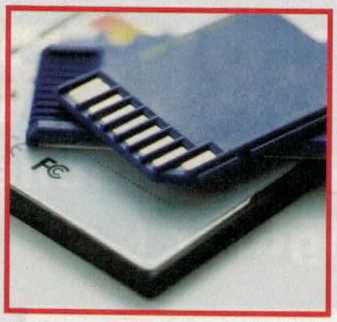
La mise en œuvre de Linux embarqué (SnapGear) sur le processeur LEON 3 est véritablement une offre de *codesign* complète pour un développement conjoint matériel et logiciel !

Auteurs : Ahmed Ben Atitallah avec la participation de Patrice Kadionik

Liens

- **[1]** Le site d'Altera : http://www.altera.com
- **[2]** Le site de Xilinx : http://www.xilinx.com
- **[3]** Le site de P. Kadionik : http://www.enseirb.fr/~kadionik/
- **[4]** Le site de P. Nouel : http://www.enseirb.fr/~nouel/
- **[5]** Le site des blocs IP libres : http://www.opencores.org
- **[6]** Le processeur LEON : http://www.gaisler.com/index.html
- **[7]** Le processeur OpenRisc : http://www.opencores.org/projects.cgi/web/or1k/overview
- **[8]** Le processeur F-CPU : http://www.f-cpu.org
- **[9]** Le processeur LatticeMico32 Open : http://www.latticesemi.com/
- **[10]** Le forum du processeur LEON : http://tech.groups.yahoo.com/group/leon_sparc/
- **[11]** *Gaisler Research : GRMON User's Manual*
- **[12]** *Gaisler Research : GRLIB IP Library User's Manual*
- **[13]** *Gaisler Research : SnapGear Linux for LEON*
- **[14]** Souissi (N.), Ben Atitallah (A.), Ghozzi (F.), Ben Ayed (M.), Masmoudi (N.), « Etude comparative de deux processeurs softcore NIOS II et LEON 3 », JTEA'06, Hammamet, Tunisie, 12-14 Mai 2006
- **[15]** Kadionik (P.), Ben Atitallah (A.), Nouel (P.), Levi (H.), « De l'usage des processeurs softcore et de Linux pour la conception des systèmes embarqués », CETSIS '07, Bordeaux, France, 29-31 Octobre 2007

Étude d'un système d'exploitation pour (TI MSP430) : pilote pour le stockage



Nous proposons d'aborder le developpement sur microcontrôleur selon un aspect « environnement exécutif » fourni par TinyOS. La plate-forme minimaliste – le microcontrôleur MSP430F149 avec 2 KB de RAM – est exploitée au mieux grâce à cet environnement de développement. Nous démontrons la mise en œuvre de cette plate-forme, munie d'une carte SD, pour le stockage de masse non volatil de données acquises périodiquement sur support formaté accessible depuis la majorité des ordinateurs personnels (FAT).

Auteurs

- Gwenhaël Goavec-Merou, Jean-Michel Friedt

1 INTRODUCTION

Deux grandes tendances se dessinent sur l'évolution de l'informatique : d'une part des plates-formes de calcul puissantes, aux ressources quasiment illimitées, mais gourmandes en énergie. D'autre part, des applications aux puissances de calcul et mémoire réduites répondant aux besoins de l'utilisateur nomade qui veut pouvoir exploiter son système informatique en tous lieux et en toutes circonstances. La contrainte énergétique décrite précédemment, due à l'autonomie de ces dispositifs, implique l'utilisation de puissances de calcul réduites entraînant ainsi un développement de logiciels contraint par la capacité de la mémoire et par la vitesse d'exécution.

Les systèmes d'exploitation (*operating systems*, OS) développés pour les ordinateurs actuels requièrent trop de ressources mémoire et de calcul, et ne constituent pas une solution pour l'adaptation à des plates-formes ayant des ressources limitées. Par ailleurs, une programmation bas niveau limite la portabilité d'algorithmes complexes sur des systèmes informatiques à caractéristiques différentes.

La mode actuelle est aux réseaux de capteur, dont chaque élément vise à une autonomie de plusieurs jours voire plusieurs années. Ces dispositifs sont déployés pour collecter de façon autonome des informations sur leur environnement, qu'il s'agisse d'études dans le cadre de comportements animaliers **[1, 2]**, climatique **[3]** ou de contrôle industriel **[4]**.

Les données acquises par ces systèmes doivent pouvoir être envoyées par une communication radio ou conservées en vue d'une exploitation ultérieure :

- la première approche fournit les informations en temps réel, mais n'est pas toujours adaptée, d'une part par manque de fiabilité de la liaison radiofréquence et d'autre part du fait de la consommation énergétique excessive de tous les modes de communication radiofréquence ;
- la seconde approche offre une plus grande sécurité du point de vue de la conservation de l'information. Le volume de données n'est limité que par le support et permet à un utilisateur de récupérer rapidement l'intégralité des informations accumulées sous réserve de disposer d'un format de stockage compatible avec son ordinateur. Cependant, l'utilisateur doit venir physiquement sur site récupérer les informations accumulées et ne sait qu'en fin d'expérience si l'acquisition a fonctionné.

Nous nous sommes donc proposés d'évaluer une implémentation portable d'un système de fichiers en vue de la conservation d'informations sur support de stockage non volatil, en s'imposant de respecter les contraintes (mémoire, puissance de calcul, consommation) d'un système fortement embarqué.

2 POURQUOI UN OS SUR MICROCONTRÔLEUR ?

La méthode de développement classiquement utilisée sur système embarqué consiste généralement en la rédaction d'une application monolithique répondant aux besoins d'une tâche dédiée. Cette application peut éventuellement commuter entre divers états si la tâche le nécessite, commutation soit contrôlée par l'obtention d'un résultat (la planète est

microcontrôleur faible consommation de masse au format FAT sur carte SD

atteinte, il faut amorcer la séquence d'atterrissage) ou de façon périodique sous le contrôle d'une horloge. Chaque nouvelle application réinvente donc son propre séquenceur de tâches (*scheduler*) répondant au mieux à ses besoins, en général sans changement de contexte afin d'économiser les ressources disponibles. L'abstraction est inexistante puisque l'application dédiée est fortement liée au matériel. Dans le meilleur des cas, le développeur optimise les performances en programmant en assembleur, sinon il se contente du C ou d'un autre langage de plus haut niveau.

Sans aller jusqu'aux multiples couches d'abstraction des systèmes d'exploitation modernes - et des ressources consommées en conséquence - une tentative de portabilité des codes pour systèmes embarqués est visée avec les environnements exécutifs : sans fournir toutes les fonctionnalités d'un système d'exploitation moderne (chargement dynamique d'exécutables et de bibliothèques partagées, gestion de la mémoire, protection de l'espace noyau distinct de l'espace utilisateur, multiplicité d'utilisateurs, shell interactif), il s'agit d'un programme monolithique développé au moyen d'outils permettant au développeur de se croire sur un système d'exploitation. Le compilateur se charge alors d'assembler les divers éléments du programme afin de générer un unique exécutable contenant toutes les fonctionnalités de l'application. TinyOS - qui a abusivement pris l'extension d'OS - suit ce mode de fonctionnement, qui n'est pas sans rappeler la distinction entre BusyBox et l'ensemble des outils GNU disponibles dans `/bin` d'un système Un*x.

3 TINYOS

L'objectif des environnements exécutifs est donc de fournir une couche d'abstraction au moment du développement, avec la possibilité d'écrire du code portable et réutilisable, sans perdre de performance au moment de l'exécution puisque l'application monolithique exploite au mieux les ressources disponibles en étant statique.

TinyOS est un ensemble de routines mises à la disposition du programmeur en vue de simplifier le développement. Le code, écrit dans un langage dont la syntaxe s'inspire du C, nesC (section 5), est interprété pour générer un code C qui est compilé.

L'intérêt de TinyOS ne réside pas spécialement dans l'application installée sur le capteur, mais au niveau de la simplification du développement. Il apporte la même notion d'abstraction qu'un système d'exploitation classique, sans imposer d'une part une trop forte dépendance entre le matériel et le logiciel, rendant dès lors possible une portabilité quasi transparente sur une multiplicité de plates-formes différentes, et d'autre part sans que le programmeur n'ait un réel besoin de connaître les spécifications du matériel sur lequel il travaille. Dans le cas d'un MSP430, l'ensemble de l'implémentation des caractéristiques d'accès au microcontrôleur est déjà réalisé. Il ne reste donc qu'à se concentrer sur l'application ou le pilote à réaliser en utilisant les routines mises à la disposition de l'utilisateur.

TinyOS offre également un certain nombre de mécanismes permettant la communication d'une manière générique par rapport au médium sous-jacent, rendant ainsi la communication homogène sur un ensemble de nœuds d'origines et de caractéristiques différentes.

Afin de se familiariser au mieux avec toutes les notions de TinyOS, nous nous sommes proposés de travailler sur

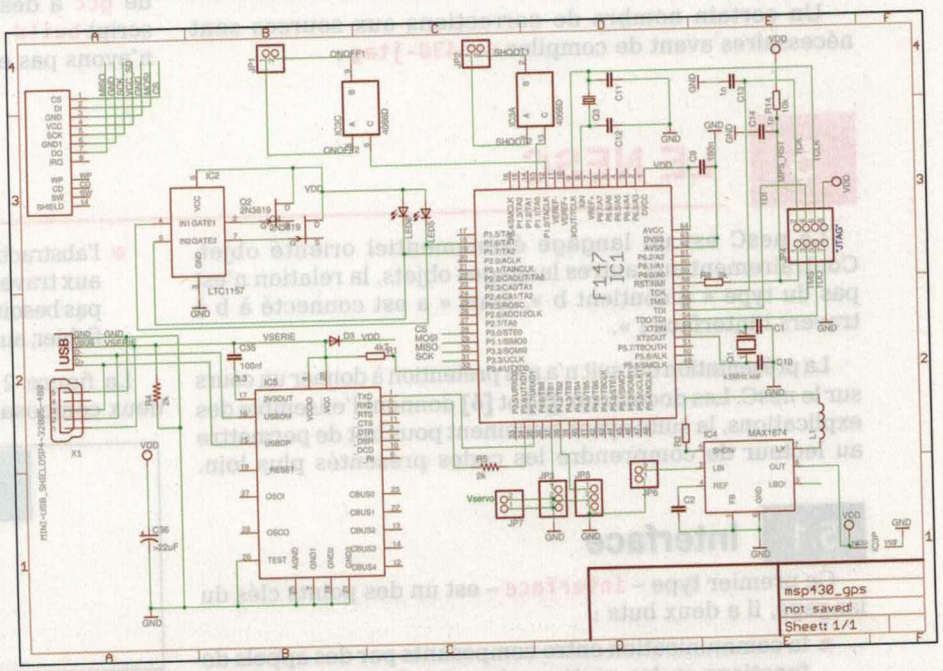


Figure 1 : Schéma de la plate-forme

un microcontrôleur supporté par cet environnement (le MSP430F149), mais sur un circuit spécifiquement développé pour une application permettant l'acquisition de données d'un récepteur GPS afin d'être stockées sur une carte mémoire utilisant un système de fichiers compatible avec n'importe quel ordinateur.

La plate-forme (Fig. 1)

- est à base d'un MSP430F149 disposant de 60 KB de mémoire flash et de 2 KB de RAM ;
- est équipée d'un récepteur GPS ET312 et de son antenne ;
- offre une communication série asynchrone en écriture seule, convertie en USB par un FT232 pour être compatible avec la plupart des ordinateurs récents ;
- fournit un emplacement pour une carte mémoire de type *secure digital* (SD) pour une communication par protocole série synchrone (SPI) ;
- est alimentée sur 4 accumulateurs NiMH pour une tension d'alimentation de 4,8 V et un régulateur linéaire LE33CZ vers 3,3 V, ou une paire d'accumulateurs NiMH et une pompe de charge MAX1674 élevant la tension à 3,3 V.

4 INSTALLATION DES OUTILS DE DÉVELOPPEMENT

Deux outils de développement seront nécessaires pour exploiter les programmes développés sous TinyOS-2.x : un cross-compilateur à destination de l'architecture cible – ici le MSP430 - et un outil de programmation pour transférer le programme compilé au microcontrôleur.

Le MSP430 se programme au travers d'une interface de communication synchrone JTAG[1]. Une solution peu coûteuse est l'utilisation d'une interface sur port parallèle : le protocole est implémenté par l'outil **msp430-jtag** dont l'installation est développée ci-dessous.

L'archive des sources s'obtient par CVS, par :

```
export CVSROOT=:pserver:anonymous@mspgcc.cvs.sourceforge.net:/cvsroot/
mspgcc
export CVS_RSH=ssh
cvs login
cvs checkout jtag
cvs checkout python
```

Un certain nombre de corrections aux sources sont nécessaires avant de compiler **msp430-jtag** :

- dans **jtag/funclets/makefile** : remplacer **msp430x2121** par **msp430x149** afin d'adapter à la version du microcontrôleur exploité dans nos applications ;
- remplacer **ASFLAGS = -mmcu=${CPU}** par **ASFLAGS = -mmcu=${CPU} -D_GNU_ASSEMBLER_** dans **jtag/funclets/eraseFlashSegment.S** ;
- dans **lockSegmentA.S**, remplacer toutes les occurrences de **LOCKA** par **0x0040**.

Nous avons toujours travaillé sur MSP430F149 au moyen de **gcc-3.3**, compilé pour générer un binaire à destination du MSP430. Cependant, afin d'utiliser une version du microcontrôleur compatible broche à broche mais fournissant plus de RAM, nous avons expérimenté l'utilisation du MSP430F1611. Ce nouveau microcontrôleur ne semble pas être supporté par **gcc-3.3**, et une version antérieure est nécessaire : mieux vaut donc tout de suite compiler **gcc-3.2**, qui supporte toutes les versions de MSP430 exploitées dans ce document. La cross-compilation de **gcc** à destination du MSP430 se fait en exécutant le script **build-gcc** fourni dans l'archive de tinyOS-1.x. Nous n'avons pas exploré l'utilisation de **gcc-4.x**.

5 LE NESC

Le nesC est un langage événementiel orienté objet. Contrairement aux autres langages objets, la relation n'est pas du type « a contient b », mais « a est connecté à b à travers l'interface i ».

La présentation qui suit n'a pas prétention à donner un cours sur le nesC. Les documents **[5]** et **[6]** donnent l'ensemble des explications, la suite ayant seulement pour but de permettre au lecteur de comprendre les codes présentés plus loin.

5.1 Interface

Ce premier type – **interface** – est un des points clés du langage, il a deux buts :

- la communication entre composants par des appels de fonctions et des gestionnaires d'événements ;
- l'abstraction logicielle. Comme les communications se font aux travers d'interfaces génériques, un composant A n'a pas besoin de savoir s'il fait un **write** (par exemple) sur un fichier, au travers d'une communication série ou par radio.

La figure 2 présente la relation qui peut exister entre deux composants.

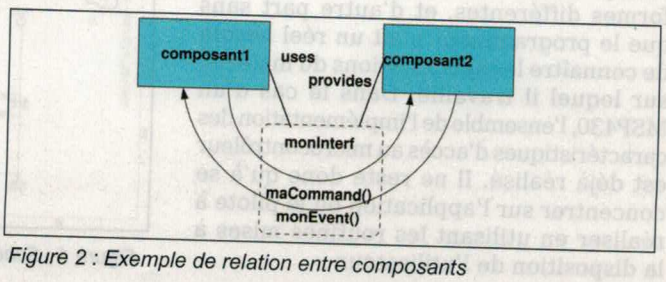


Figure 2 : Exemple de relation entre composants

Le principe de l'interface est le même que dans d'autres langages, tels que C++ ou Java, c'est-à-dire imposer un ensemble de fonctions qu'un composant devra implémenter dès lors qu'il déclare sa relation avec l'interface. Toutefois, pour permettre l'aspect événementiel, elle ajoute également une notion de bidirectionnel. Ceci est possible en proposant des fonctions allant de l'utilisateur vers le fournisseur et d'autres allant du fournisseur vers l'utilisateur.

La figure 3 donne la représentation d'une interface simple.

```
interface interf1 {
        command void maCommand();
        event void monEvent();
}
```

Figure 3 : Exemple de définition d'une interface

Afin de pouvoir utiliser l'interface, dans un mode ou dans l'autre, le composant aura besoin de préciser ses intentions :

- avec **uses** pour en faire usage ;
- avec **provides** pour en être fournisseur.

Comme présenté dans la figure 3, il apparaît deux types de fonctions :

- celles commençant par **command**, qui doivent être implémentées par le composant fournissant l'interface ;
- celles commençant par **event**, devant l'être par le composant utilisateur.

Les fonctions issues d'une interface donnée ne sont accessibles qu'en utilisant le nom pleinement défini de celle-ci :

```
nom_interface.nom_fonction(arguments);
```

C'est sur cette caractéristique que se base toute la notion d'abstraction de TinyOS.

Lors de l'utilisation et selon le type de la fonction, il faudra utiliser **call nom_interface.nom_fonction(arguments);** pour une **command** et **signal nom_interface.nom_fonction(arguments);** pour utiliser un **event**.

Pour la définition des fonctions, la même règle est à respecter. Ainsi, la signature de la définition d'une fonction se fera par **type_fonction type_retour nom_interface.nom_fonction(arguments)** et l'appel (*call* ou signal) se fera par **type_appel nom_interface.nom_fonction(arguments);**.

5.2 Structure des composants

Structurellement, un fichier d'entité est composé de deux zones, comme dans le cas de l'ADA ou du VHDL. Une première définit la vue externe et une seconde, selon le type, donne l'implémentation du module ou une mise en relation de composants.

5.2.1 Vue externe

Pour ne pas risquer de confusion, cette partie ne sera jamais appelée interface, même si son rôle est sensiblement le même que ce que l'on trouve dans un fichier **.h** en C, par exemple.

Elle définit le type de l'entité ainsi que les entrées/sorties de celle-ci, au travers des interfaces.

Les entrées/sorties se font en définissant quelles sont les interfaces qui sont utilisées et selon quelle relation.

```
type ent1 {
        provides {
                interface interf1;
        }
        uses interface interf2;
}
implementation {
                ...
}
```

Figure 4 : Exemple de fichier d'entité

La figure 4 présente la définition de la vue externe pour **ent1**. L'interface **interf1** (Fig. 3) est fournie, ainsi un autre composant en relation avec celui-ci pourra faire des appels à **maCommand**, mais devra également définir une implémentation pour **monEvent**.

Notez que les accolades après **provides** peuvent être omises si une seule interface est définie, comme montré pour **interf2**. Il est également possible de définir un alias pour une interface afin de la rendre plus claire ou d'éviter un conflit dans le cas où un composant ferait usage ou implémenterait plusieurs interfaces du même type.

5.3 Type d'une entité

Il existe en nesC deux types d'entités :

- les modules dont la partie implémentation contient la définition des **commands** ou des **events** ;
- les configurations qui ne contiennent pas de code, mais permettent le branchement de modules entre eux.

Si l'on fait un parallèle avec un circuit physique, le module correspond à un composant et la configuration au circuit imprimé.

La norme dit que le fichier de configuration pour un module aura un nom finissant par un C (**monModuleC.nc**) et le fichier d'implémentation (type **module**) aura un nom finissant par un P (**monModuleP.nc**).

5.3.1 Type configuration

Il permet de brancher plusieurs modules entre eux. En prenant l'exemple d'un module pour un traitement quelconque, la configuration de celui-ci permettrait de pouvoir :

- fournir les ressources (interfaces) pour l'utilisation de ce module ;
- réaliser les raccordements sur les couches inférieures d'une manière non visible pour les modules faisant usage de celui-ci.

```
configuration maConfig {
        provides interface interf1;
}
implementation {
        components PlatformPlop;
        components monDriver;
        interf1 =  monDriver.interf1;
        monDriver.stdControl -> PlatformPlop.stdControl;
}
```

Figure 5 : Fichier de configuration d'un module

Dans la figure 5, deux types de relations sont présentées :

- celle liée par un `=` fait transiter l'information si elle a le même type (*uses* ou *provides*),
- celle liée par un `->` pour lier l'interface d'un module utilisant celle-ci à l'interface d'un module la fournissant. La flèche va toujours de l'utilisateur vers le fournisseur.

Dans le cas présenté, un utilisateur de `maConfig` ne connaîtra que l'interface `interf1`, les relations entre `monDriver` et `PlatformPlop` étant cachées.

5.3.2 Le type module

Il permet de fournir l'implémentation du module en lui-même. Celui-ci va contenir :

- l'implémentation des `commands`/`events` définis dans les interfaces qu'il uses/provides ;
- des variables locales au module ;
- des fonctions elles aussi locales ;
- des *tasks*, qui sont des fonctions appelées d'une manière asynchrone et dont la portée est limitée au module.

6 PLATE-FORME

La création d'une plate-forme n'est nécessaire que lorsque la définition du circuit cible n'est pas fournie d'origine par TinyOS. Cependant, étant donné que les applications se compilent en spécifiant la plate-forme cible, il s'agit de l'étape obligatoire avant de pouvoir développer une application sur un nouvel environnement matériel.

La plate-forme est une description du matériel fournissant par la suite l'abstraction entre les applications et la partie matérielle. L'explication de la création d'une plate-forme se fera au travers de la réalisation de celle de l'architecture de la figure 1 et se nommera `projet`, en suivant le tutoriel disponible à **[7]**.

L'ensemble des fichiers à créer/modifier/utiliser se trouvent dans le répertoire racine des sources de TinyOS, celui-ci dépend de la méthode d'installation. Le chemin est défini par la variable `TOSROOT`. Dans la suite de ce document, les fichiers et répertoires seront toujours en chemin relatif vis-à-vis de cette racine.

6.1 Définition

Une plate-forme permet une couche d'abstraction entre les applications et la carte utilisée. En d'autres termes, pour une communication selon un protocole quelconque, la hiérarchie du système se compose de la manière suivante :

1. L'application utilise le type correspondant sans toutefois connaître son implémentation par le système, ni le câblage sur la carte. Il n'a besoin que de savoir les actions possibles.
2. Le module ne connaît pas non plus le câblage. Il se contente de faire les traitements liés au module et passe les « ordres » au fichier que le créateur de la plate-forme a réalisé.
3. Le fichier de la plate-forme définit la liaison avec les ports du microcontrôleur ainsi que les paramètres de configuration sans connaître les registres.
4. Le pilote du microcontrôleur fait au final l'opération adéquate selon le besoin.

6.2 Structure minimale d'une plate-forme

Celle-ci est définie en deux entités principales qui sont :

1. Un fichier se trouvant dans `support/make`. Il est de la forme `NomPlatform.target`. Ce fichier informe la commande `make` de l'existence de la plate-forme.
2. Un répertoire se trouvant dans `tos/platforms`, qui va contenir l'ensemble des fichiers de configuration de la plate-forme. Chaque module s'attend à trouver un fichier portant un nom bien spécifique, permettant de simplifier le travail d'intégration.

6.3 Mise en place de la nouvelle plate-forme

En tout premier lieu, il est nécessaire de créer un répertoire projet dans `tos/platforms`, avec la commande `mkdir projet`.

1. `.projet.target` : dans le répertoire `support/make`, nous créons un fichier `projet.target`, qui va contenir les lignes présentes dans le listing 1 :

```
PLATFORM = projet
$(call TOSMake_include_platform,msp)

projet: $(BUILD_DEPS)
	@:
```

Listing 1 : projet.target

La première ligne précise le nom de la plate-forme. Les lignes suivantes donnent les informations dynamiques sur les dépendances de la plate-forme ainsi que d'autres règles de compilation liées au microcontrôleur.

2. `.platform` : pour la suite, tout se déroulera dans `tos/platforms/projet`. `.platform` concerne les spécifications sur les répertoires contenant les bibliothèques de fonctions et les options de compilation. Contrairement à un logiciel pour un système d'exploitation « classique », où les configurations et informations passées au compilateur sont fournies avec les sources

de l'application, ces informations sont, dans TinyOS, au niveau de la plate-forme.

```
push( @includes, qw(

%T/chips/msp430
%T/chips/msp430/adc12
%T/chips/msp430/dma
%T/chips/msp430/pins
%T/chips/msp430/timer
%T/chips/msp430/usart
%T/chips/msp430/sensors
%T/lib/timer
%T/lib/serial
%T/lib/power
));

push ( @opts, qw(
-gcc=msp430-gcc
-mmcu=msp430x149
-fnesc-target=msp430
-fnesc-no-debug
-fnesc-scheduler=TinySchedulerC,TinySchedulerC.TaskBasic,TaskBasi
 c,TaskBasic,runTask,postTask
));
```

Listing 2 : .platform

Le listing 2 contient les configurations dans le cas de la plate-forme **projet**. Il est écrit en PERL et contient les deux parties suivantes :

- a. les répertoires qui contiennent les bibliothèques ;
- b. les options qui seront passées au compilateur.

3. **hardware.h** : ce fichier est inclus lors de la compilation d'une application. Il sert à définir des constantes et des en-têtes (*headers*).

4. **PlatformC.nc** : ce fichier ne sert qu'à fournir une implémentation de l'Init, qui sera appelée automatiquement lors du démarrage de la plate-forme. Il permet de fournir un comportement automatique à ce moment, avant le lancement de l'application elle-même.

```
#include "hardware.h"

configuration PlatformC {
  provides interface Init;
}
implementation {
  components PlatformP, Msp430ClockC;
  Init = PlatformP;
  PlatformP.Msp430ClockInit -> Msp430ClockC.Init;
}
```

Listing 3 : PlatformC

5. **PlatformP.nc** : ce fichier contient l'initialisation des divers composants qui seront utilisés dans cette plate-forme.

```
#include "hardware.h"

module PlatformP {
  provides interface Init;
  uses interface Init as Msp430ClockInit;
  uses interface Init as LedsInit;
}
implementation {
  command error_t Init.init() {
    call Msp430ClockInit.init();
    call LedsInit.init();
    return SUCCESS;
  }
  default command error_t LedsInit.init() { return SUCCESS; }
}
```

Listing 4 : PlatformP

PlatformP implémente l'interface Init pour lancer l'initialisation de l'horloge interne du MSP430 ainsi que l'initialisation de LED.

6.4 Exemple pratique : contrôle des ports numériques

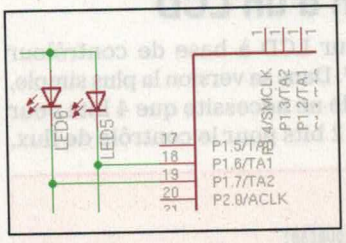


Figure 6 : Câblage des LED

Les ports d'entrée/sortie numériques (*General Purpose Input-Output*, GPIO) sont probablement les périphériques les plus simples d'accès : nous allons les exploiter pour contrôler l'allumage et l'extinction de LED (Figure 6), puis pour afficher du texte sur un afficheur LCD compatible HD44780 (Figure 9).

La figure 6 présente le câblage physique de deux LED de la carte aux bornes du microcontrôleur. Au niveau de leur implémentation dans TinyOS, il faut créer le fichier suivant, imposé par **LedsC.nc** :

```
#include "hardware.h"
configuration PlatformLedsC {
  provides interface GeneralIO as Led0;
  provides interface GeneralIO as Led1;
  provides interface GeneralIO as Led2;
  uses interface Init;
}
implementation {
  components HplMsp430GeneralIOC as GeneralIOC,
    new Msp430GpioC() as Led0Impl,
      new Msp430GpioC() as Led1Impl;

  components new NoPinC() as Led2Impl;
  components PlatformP;

  Init = PlatformP.LedsInit; // Raccorde l'event init à celui de
PlatformP

  Led0 = Led0Impl;
  Led0Impl -> GeneralIOC.Port16;
  Led1 = Led1Impl;
  Led1Impl -> GeneralIOC.Port17;
  Led2 = Led2Impl; // No led2 on board
}
```

Listing 5 : PlatformLedsC

Cette configuration permet de préciser le branchement matériel pour chaque LED définie par le pilote TinyOS. Les 5 dernières lignes définissent que led0 est raccordé au Port16 (noté P1.6 dans la documentation (*datasheet*) du MSP430) et led1 est raccordé au Port17 (noté P1.7). Le cas de led2 est particulier car le pilote Leds définit une troisième LED absente du montage physique. Celle-ci est connectée sur NoPinC signifiant son absence.

Un exemple d'utilisation des LED est l'application `apps/Blink`, qui peut fonctionner sans avoir besoin de la moindre modification.

Note

Dans le cas d'une plate-forme exploitant moins de 3 LED, il faut utiliser NoPinC, comme présenté plus haut, pour désactiver les LED absentes. Dans le cas où la carte contiendrait plus de 3 LED, il faut copier les fichiers `tos/system/LedsC.nc` et `tos/system/LedsP.nc` afin d'ajouter les LED manquantes dans le pilote officiel.

6.5 Exploitation d'un LCD

Le protocole d'un afficheur LCD à base de contrôleur HD44780 est bien documenté[2]. Dans sa version la plus simple, l'implémentation du protocole ne nécessite que 4 bits pour les données à transmettre et 2 bits pour le contrôle de flux.

```
#include "hardware.h"
configuration PlatformLcdC {
  provides interface GeneralIO as LcdData0;
  provides interface GeneralIO as LcdData1;
  provides interface GeneralIO as LcdData2;
  provides interface GeneralIO as LcdData3;
  provides interface GeneralIO as LcdE;
  provides interface GeneralIO as LcdRS;
  uses interface Init;
}
implementation {
  components Hpl430GeneralIOC as GeneralIOC,
    new Msp430GpioC() as LcdData0Impl,
    new Msp430GpioC() as LcdData1Impl,
    new Msp430GpioC() as LcdData2Impl,
    new Msp430GpioC() as LcdData3Impl,
    new Msp430GpioC() as LcdEImpl,
    new Msp430GpioC() as LcdRSImpl;

  components PlatformP;

  Init = PlatformP.LcdInit;

  LcdData0 = LcdData0Impl;
  LcdData0Impl -> GeneralIOC.Port24;

  LcdData1 = LcdData1Impl;
  LcdData1Impl -> GeneralIOC.Port25;

  LcdData2 = LcdData2Impl;
  LcdData2Impl -> GeneralIOC.Port26;

  LcdData3 = LcdData3Impl;
  LcdData3Impl -> GeneralIOC.Port27;

  LcdE = LcdEImpl;
  LcdEImpl -> GeneralIOC.Port23;

  LcdRS = LcdRSImpl;
  LcdRSImpl -> GeneralIOC.Port22;

}
```

Listing 6 : PlatformLcdC

Le listing 6 montre le fichier de configuration. Celui-ci est globalement équivalent à celui des LED, si ce n'est par le volume de broches défini. En effet, avec TinyOS, il n'est pas possible d'accéder à un port complet, mais uniquement aux broches individuelles. Il utilise le même type de ressources permettant ainsi d'émuler l'écriture des données sur le port.

De ce fait, l'écriture sur le port se fait de façon un peu fastidieuse par le code proposé dans le listing 7.

```
// Ecrit un demi octet
void writeDB(uint8_t val) {
    val = val & 0x0f;
    if (val & LCD_DATA0)
      call LcdData0.set();
    else call LcdData0.clr();

    if (val & LCD_DATA1)
      call LcdData1.set();
    else call LcdData1.clr();

    if (val & LCD_DATA2)
      call LcdData2.set();
    else call LcdData2.clr();

    if (val & LCD_DATA3)
      call LcdData3.set();
    else call LcdData3.clr();
}
```

Figure 7: Fonction écrivant un demi octet

Il n'existe pas d'application dans TinyOS pour tester le bon fonctionnement du module : il est donc nécessaire d'en ajouter une. En reprenant l'application `Blink`, il suffit de remplacer le contenu de Boot.booted par le contenu du listing 8.

```
event void Boot.booted() {
    call Lcd.write("Hello World",11);
}
```

Figure 8 : Écriture de Hello World sur le LCD

La fonction `Lcd.write()`, définie dans l'interface Lcd, va gérer tout le protocole du Lcd en vue de l'affichage et faire autant d'appels à `writeDB` qu'il y a de caractères à écrire.

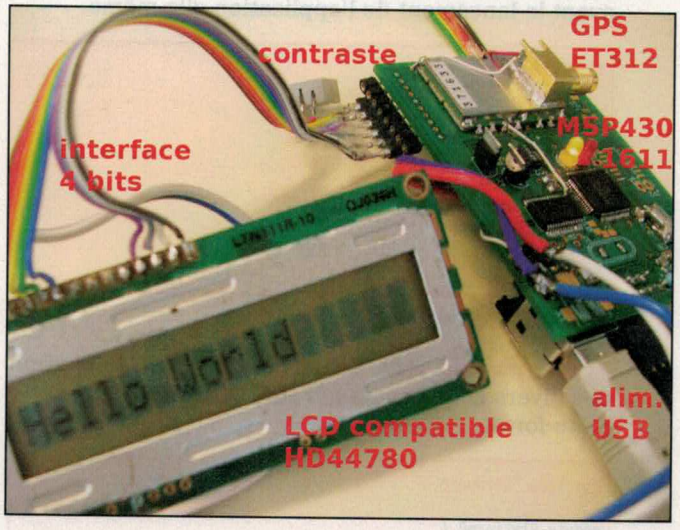


Figure 9 : Circuit comprenant un MSP430F1611 auquel sont connectées deux LED, un écran LCD compatible HD44780 en mode de communication 4 bits, un récepteur GPS ET312 et un support de carte SD, le tout alimenté par le bus USB. Dans cet exemple, l'affichage du message de démonstration du bon fonctionnement de l'écran LCD qui servira au débogage des diverses applications qui vont suivre.

7 COMMUNICATIONS NUMÉRIQUES

7.1 Communication asynchrone RS232

Ce protocole de communication asynchrone - normalisé sous la nomenclature RS232 - propose une communication bidirectionnelle nécessitant 2 fils (transmission et réception de données), supposant que les deux interlocuteurs possèdent chacun une horloge de fréquence connue. Le seul moyen de synchroniser les deux interlocuteurs est d'une part un échange au préalable du protocole de communication (vitesse et nombre de bits/octets transmis), et au cours de la transmission, deux transitions nommées *start* et *stop bits* annonçant les début et fin de communication. Ce protocole est le plus couramment utilisé de par sa simplicité : il est donc implémenté sous forme matérielle (*Universal Asynchronous Receiver/Transmitter*, UART) dans la majorité des microcontrôleurs, et ce sera notre premier exemple de développement sous TinyOS.

TinyOS offre un mécanisme de communication intégré s'apparentant aux messages TCP/IP classiques : les paquets sont encapsulés dans une trame contenant un certain nombre d'informations. Cela permet leur routage et la vérification du contenu, par somme de redondance cyclique. L'intérêt est d'offrir, d'une manière transparente, une interface simple cachant le mode de communication exploité (radio, zigbee, bluetooth, série), compatible quel que soit l'endianess du microcontrôleur.

Le modèle TinyOS possède clairement des avantages pour l'application visée - les réseaux de capteurs - car il permet d'homogénéiser la communication et d'en assurer la fiabilité. Mais cette encapsulation de la communication est un handicap pour les outils classiques tels que **minicom** ou avec un périphérique exploitant une connexion série. Par exemple, dans le cas d'un récepteur GPS, qui fournit ses informations au moyen d'une chaîne ASCII, le modèle TinyOS n'est pas exploitable. L'encapsulation des paquets proposée par TinyOS doit donc être évitée dans le cadre du projet qui nous intéresse, afin que TinyOS ne rejette pas des paquets ne se conformant pas à son modèle de communication.

Afin de pouvoir exploiter l'UART du microcontrôleur, **SerialActiveMessage** utilise deux fichiers spécifiques à la plate-forme cible : **PlatformSerialC.nc** et **PlatformSerialP.nc**. Ces deux fichiers permettent :

- de relier le pilote et la couche matérielle ;
- de spécifier la configuration du port.

Une exploitation brute de l'UART (sans encapsulation des données) va donc consister à définir un module pour la réception et l'envoi de données, directement raccordé sur la plate-forme et n'utilisant pas la couche de communication fournie par TinyOS.

Le listing suivant donne l'interface publique raccordant le pilote et le microcontrôleur :

```
#include "hardware.h"
configuration PlatformSerialC {
  provides interface StdControl;
  provides interface UartStream;
  provides interface UartByte;
}
implementation {

  components new Msp430Uart1C() as UartC, projetSerialP;
  UartStream = UartC.UartStream;
  UartByte = UartC.UartByte;
  StdControl = projetSerialP.Control;
  projetSerialP.Msp430UartConfigure <- UartC.Msp430UartConfigure;
  projetSerialP.Resource -> UartC.Resource;
  projetSerialP.ResourceRequested -> UartC.ResourceRequested;
  components LedsC;
  projetSerialP.Leds -> LedsC;
}
```

Figure 10 : PlatformSerialC

L'interface la plus importante à prendre en compte pour la suite est **UartStream**, qui offre principalement :

- une commande permettant l'envoi d'une chaîne de caractères : **async command error_t send(uint8_t* buf, uint16_t len);** ;
- un événement qui permettra au programme appelant d'être averti de la fin de l'envoi, ainsi que de son succès ou de son échec : **async event void sendDone(uint8_t* buf, uint16_t len, error_t error);** ;
- un événement qui est généré à chaque fois qu'un octet arrive sur l'UART du microcontrôleur : **async event void receivedByte(uint8_t byte);**.

Le fichier **PlatformSerialP.nc** est utilisé principalement pour la gestion du lancement et de l'arrêt de l'UART. Il contient une structure définissant la configuration du port en termes de vitesse de transfert, de type et fréquence d'horloge, de parité, etc. :

```
msp430_uart_union_config_t msp430_uart_proj_config = { {
  ubr: UBR_32KHZ_4800,
  umctl: UMCTL_32KHZ_4800,
  ssel: 0x01,
  pena: 0,
  pev: 0,
  spb: 0,
  clen: 1,
  listen: 1,
  mm: 0,
  ckpl: 0,
  urxse: 0,
  urxeie: 1,
  urxwie: 0,
  urxe: 0,
  utxe: 1} };
```

Figure 11 : Structure de configuration de la communication série

Une fois ces fichiers à disposition et raccordés à l'application ou à un pilote, il est possible d'obtenir des données telles que des trames NMEA fournies par un récepteur GPS.

7.2 Acquisition des trames d'un récepteur GPS

Le récepteur GPS ET312 équipant la carte utilise le protocole RS232 au débit de 4800 bauds, protocole N81. Les mêmes broches sont exploitées pour la communication série synchrone et asynchrone (Tx pour l'émission de messages vers un ordinateur, Rx pour la réception des données du GPS). Le second port de communication disponible sur le MSP430 sera utilisé plus tard pour la liaison synchrone de type SPI.

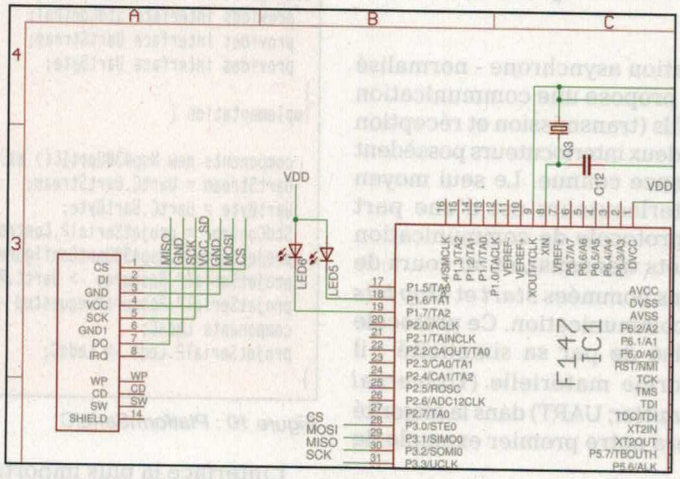


Figure 12 : Câblage de la carte SD

Les trames GPS sont des chaînes de caractères en ASCII directement compréhensibles, commençant par un `$` et se finissant par un `CRLN`. Comme TinyOS gère les interruptions, il est possible d'être averti de la réception d'un octet sur l'UART afin de le traiter et le stocker. Afin de ne pas risquer de pertes liées à l'aspect asynchrone, le module reçoit un tableau et sa taille. Il avertit l'application lorsque le tableau est complètement rempli.

Le démarrage de l'acquisition se fait grâce à l'interface `ReadStream`. `SplitControl` sert à la gestion de l'UART. Le fonctionnement du module est donc ainsi :

- Lorsque `ReadStream.postBuffer(...)` n'a pas été appelé, l'état est `GPS_IDLE`. Dans cet état, le module ne s'occupe pas des octets qu'il reçoit.
- Après un appel à `ReadStream.postBuffer`, le module se met dans l'état `GPS_READ`. À partir de ce moment, il va stocker tous les caractères reçus.
- Lorsque le tableau est rempli, il se remet dans l'état `GPS_IDLE` et signale à l'application qu'il a fini sa lecture.

Afin de tester ce pilote, une première application simple a été mise en œuvre, consistant à copier sur la sortie du port série (lié à un PC) les trames lues sur l'entrée du port série lié au récepteur GPS. Comme les deux communications ont la même vitesse, un tampon correspondant à la longueur d'une trame permet de rendre les deux actions indépendantes.

Le stockage des trames reçues sur le port série de l'ordinateur permet le calcul du débit d'informations.

```
$GPGGA,062546.000,4722.9018,N,00600.0719,E,1,07,2.6,219.5,M,47.9,M,,0000*53
$GPGSA,A,3,27,29,09,12,30,02,26,,,,,,5.0,2.6,4.3*35
$GPGSV,3,1,12,30,77,322,41,05,60,287,,12,56,069,49,29,47,204,44*7F
$GPGSV,3,2,12,14,39,254,,26,31,219,42,02,30,084,48,09,22,140,36*75
$GPGSV,3,3,12,31,20,310,17,27,17,140,36,04,17,044,,25,12,312,*7B
$GPRMC,062546.000,A,4722.9018,N,00600.0719,E,0.13,329.05,061209,,*09
```

En comptant le nombre d'octets pour un ensemble de trames avec la même estampille temporelle, il est possible de déterminer un débit de 402 octets par seconde.

7.3 Communication synchrone SPI : application à la carte mémoire SD

La communication asynchrone fournit un débit de communication limité du fait de la difficulté à synchroniser des horloges, basées sur des résonateurs de qualité médiocre, cadençant les deux systèmes numériques communiquant. L'alternative est la communication synchrone, qui partage l'horloge entre les deux dispositifs : des débits considérablement plus élevés peuvent ainsi être obtenus, au détriment de la distance de communication. Le bus SPI est une implémentation de bus de communication synchrone, asymétrique puisque distinguant la liaison du maître à l'esclave (MOSI, *Master Out Slave In*) et de l'esclave vers le maître (MISO). Un quatrième signal, *Chip Select* (CS), permet d'activer le périphérique du bus qui doit communiquer avec le maître (Fig. 12).

Le protocole SPI, rapide (quelques Mb/s) et couramment disponible sur microcontrôleurs, est implémenté de façon matérielle dans le MSP430F149 en partageant des ressources avec un port de communication asynchrone (UART). Ainsi, l'exploitation d'un bus SPI nous prive du second port de communication asynchrone.

L'implémentation matérielle du protocole **[8, chap.14]** SPI ne fournit qu'un octet de tampon (*buffer*). Contrairement au bus asynchrone tel que le RS232, toute transaction sur SPI est initiée par le maître (microcontrôleur) : il ne peut donc pas y avoir perte de données en cas de délai dans la réponse à une transaction.

L'implémentation de la communication SPI pour le MSP430 ne sera pas nécessaire, car déjà disponible dans les routines d'exploitation du microcontrôleur. La seule tâche nécessaire est de définir, à l'instar de l'UART, les configurations (en termes de source et de vitesse d'horloge, de signe de la phase, etc.) ainsi que la broche correspondant au signal d'activation du périphérique.

Ce bus local est notamment utilisé pour communiquer avec les cartes mémoire SD (et leur ancêtre *MultiMediaCard*, MMC), qui fournissent ainsi un support de stockage de masse non volatil à faible coût, ne nécessitant que peu de connexions électriques pour leur mise en œuvre. Nous allons par conséquent utiliser comme prétexte le stockage des trames GPS sur un support non volatil pour implémenter un pilote pour TinyOS, permettant de communiquer avec les cartes SD.

7.4 Support de la carte SD pour TinyOS

Il n'existe pas à l'heure actuelle de pilote TinyOS-2.x destiné à l'exploitation d'une carte SD : la seule implémentation

existante est destinée à la version 1.x. Cette solution ne répondant pas au modèle hiérarchique de TinyOS-2.x, il nous a été plus efficace de produire un nouveau pilote parfaitement intégré et homogène dans la version actuelle du système, permettant sa totale indépendance vis-à-vis du matériel.

Cette absence de support de carte SD dans TinyOS-2.x s'explique par la structure des plates-formes nativement supportées par TinyOS : celles-ci n'offrent du stockage non volatil que grâce à des composants de mémoire soudés sur la carte et ne répondant pas aux spécifications du protocole de la carte SD.

La carte SD permet de dialoguer soit à travers un protocole natif, soit à travers le protocole SPI. Les articles **[9, 10]** et le site **[11]** offrent la description et des exemples d'implémentation du protocole de communication par SPI. Il ne sera donc fait mention que des points qui sont spécifiques à l'implémentation d'un tel pilote pour TinyOS.

La carte SD offre les caractéristiques de lecture et d'écriture suivantes :

- Lecture d'un nombre d'octets allant de 1 à 512, avec pour seule contrainte que l'adresse de début et l'adresse de fin soient contenues dans le même secteur. Le nombre d'octets étant par défaut 512, il est toutefois possible d'en fixer la valeur au moment de l'initialisation ou lors de l'écriture.
- Ecriture par blocs d'une taille fixe de 512 octets (1 secteur), l'adresse de début devant être un multiple de 512.
- Effacement du contenu d'un nombre arbitraire de secteurs.

7.4.1 Connexion SPI

La première étape dans la création du pilote va consister, à l'instar du GPS, à créer les fichiers qui feront la liaison entre le pilote et le SPI du microcontrôleur. Globalement, ces fichiers sont identiques à ceux utilisés pour le GPS ou la liaison RS232 :

```
#include "hardware.h"
configuration PlatformSdC {
  provides {
    interface SplitControl;
    interface SpiByte;
  }
}
implementation {
  components projetSdP;
  SplitControl = projetSdP.Control;

  components new Msp430Spi0C() as SpiC;
  projetSdP.Msp430SpiConfigure <- SpiC.Msp430SpiConfigure;
  projetSdP.Resource -> SpiC.Resource;
  SpiByte = SpiC;
}
```

Listing 7 : PlatformSdC

Le listing 7 présente le raccordement avec le module gérant le SPI sur le MSP430 (`Msp430Spi0C`). Contrairement au GPS, le dialogue avec le matériel se fait grâce à l'interface `SpiByte`, qui n'offre qu'une seule fonction `command uint8_t write( uint8_t tx );`. Celle-ci envoie un octet et retourne la réponse du périphérique.

L'autre fichier comporte, comme pour le GPS, une structure pour la configuration de la communication (Fig. 13).

```
msp430_spi_union_config_t msp430_spi_proj_config =
  {{
    ubr: 0x0004,
    ssel: 11,
    clen: 1,
    listen: 0,
    mm: 1,
    ckph: 1,
    ckpl: 0,
    stc:1
    }};
```

Figure 13 : Configuration de l'interface SPI

7.4.2 Interface d'exploitation

La seconde étape consiste à définir les interactions entre le pilote SD et les couches supérieures. L'utilisation d'un mécanisme tel que celui mis en œuvre pour le RS232 impose que le pilote contienne un tampon de 512 octets, i.e. le nombre d'octets d'un secteur.

Cependant, la mémoire d'un microcontrôleur étant relativement faible, cet encombrement risquerait de réduire le volume de mémoire disponible pour une application utilisant ce pilote. Le module a donc été développé sur un modèle classique de fonctions ne rendant la main qu'une fois l'action finie. C'est l'application qui fournit le tampon contenant les données qui ne seront pas copiées au niveau du pilote. Ainsi, le choix du fonctionnement pour les couches exploitant la SD est laissé au soin du développeur.

L'interface se présente ainsi :

```
/**
 * SdIO
 * sert à l'exploitation de la carte sd
 * @author Gwenhaël GOAVEC-MEROU
 */

interface SdIO {
  /**
   * Commande pour la demande d'écriture d'une chaine
   * la commande est immédiate (au retour l'action est faite)
   *
   * @param addr : adresse de début d'écriture
   * @param buf tableau à envoyer
   *
   * @return SUCCESS Si la commande est acceptée
   */
  command error_t write(uint32_t addr, uint8_t*buf);

  /**
   * Commande pour la demande de lecture d'une chaine
   * la commande est immédiate (au retour l'action est faite)
   *
   * @param addr : position de début de lecture
   * @param buf : tableau dans lequel mettre l'information
   * @param count longueur du tableau
   *
   * @return SUCCESS Si la lecture est bonne
   */
  command error_t read(uint32_t addr, uint8_t*buf, uint16_t *count);
}
```

Listing 8 : interface d'utilisation du module SD

Les deux fonctions prennent l'offset (en octet) du début de la zone à écrire ou lire, le tampon contenant les données à écrire ou dans lequel seront mises les données lues, et la taille lue.

7.4.3 Mesure de débit

Afin de pouvoir obtenir une mesure de débit, une petite application simple a été réalisée. Celle-ci se contente d'écrire 2048 fois un buffer de 512 octets.

Le temps d'écriture de 1 Mo est de 1min53s, soit un débit d'environ 9 Ko/s.

8 STOCKAGE FORMATÉ EN MÉMOIRE NON VOLATILE

L'objectif de l'implémentation d'un système de fichiers dans TinyOS est d'être en mesure, sur le système embarqué, de stocker les données acquises, pour ensuite pouvoir les restituer et les exploiter à l'aide d'un ordinateur et ceci sans avoir besoin d'utiliser une fonction dédiée du système embarqué chargée de la restitution des informations.

Les raisons d'allier l'utilisation d'une carte mémoire amovible à un système de fichiers sont :

1. De réduire le temps d'arrêt du capteur, puisque la carte peut être rapidement échangée par une autre. Cette opération est plus fiable que le transfert par RS232, notamment en environnement hostile.
2. La carte sans système de fichiers peut être utilisée en RawWrite (écriture brute des données sans formatage). Cette solution impose :
   - un post traitement avant exploitation des données ;
   - que l'utilisateur ait certaines compétences afin de récupérer les données ;
   - une limite dans la possibilité de stocker des informations de diverses natures dans des emplacements séparés tels que le permet une multiplicité de fichiers.

Le choix d'un système de fichiers répond à plusieurs critères :

1. A ressources réduites, système économe, ce qui exclut d'emblée tous les systèmes journalisés, qui bien que réduisant le risque de perte de données, entraînent l'augmentation du nombre de lectures, d'écritures et donc de traitements.
2. Il faut qu'il soit multiplate-forme, afin que n'importe qui puisse nativement être en mesure de récupérer le contenu.

Le meilleur choix concerne un système de fichiers originellement développé pour des ordinateurs disponibles il y a une vingtaine d'années, dont les performances étaient proches de celles des microcontrôleurs actuels, tels que Minix, CP/M ou FAT. Le seul survivant encore largement supporté est FAT, avec diverses déclinaisons, dont la plus simple est encore compatible avec les systèmes d'exploitation les plus récents.

Remarque

Tout support de stockage, pour être compatible et exploitable sur un ordinateur, doit comporter au tout début un MBR qui fournit des informations relatives au support en lui-même et les informations sur les partitions du support. Dans le cas de l'implémentation dans TinyOS, la seule information nécessaire se trouve être le numéro du premier secteur de la partition utilisée. La formule pour trouver cette information est :

```
debPartition = (*(uint32_t   *)&buf[(446+8)+((numPart-1)*16)]);
```

446 correspond à la position de la table de partition, 8 est le décalage pour obtenir le numéro du secteur de début de partition et 16 est la taille d'une entrée de partition.

Après une présentation globale de la structure d'un support de stockage et du système de fichiers FAT16 **[12]**, nous verrons les contraintes à prendre en compte lors de son implémentation afin de le rendre polyvalent, c'est-à-dire d'avoir de bonnes performances dans le cas d'acquisitions de données où le facteur temps est important, mais également économe en énergie dans le cas d'acquisitions de petits volumes d'informations sur un délai de plusieurs mois, voire années.

La troisième partie concernera l'implémentation à proprement parler, dans laquelle nous présenterons le fonctionnement de chaque module ainsi que les codes les plus pertinents.

8.1 Support physique et partitionnement

Un support de stockage physique, tel que la SD, n'est au final qu'un grand tableau découpé en blocs plus petits, les secteurs, d'une taille de 512 octets pour la SD.

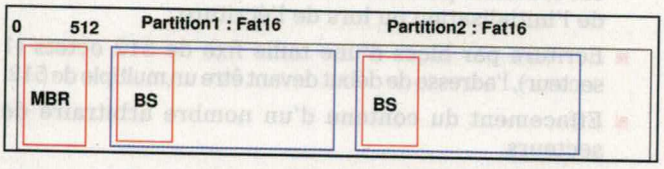


Figure 14 : Exemple de partitionnement d'un support de stockage

Afin d'utiliser un système de fichiers, le support physique doit être partitionné (avec un outil comme `fdisk`, par exemple). Ce partitionnement ajoute au tout début du support physique (adresse 0) une structure nommée MBR, ayant pour fonction de :

- fournir des informations sur le support ;
- stocker (si besoin) un exécutable permettant le boot depuis le support de stockage ;
- fournir des informations telles que la taille et l'adresse des partitions contenues.

Nous allons surtout nous intéresser au dernier point, car c'est grâce à cela qu'il est possible d'accéder à une partition.

```
000001B0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 01  ................
000001C0  01 00 06 0A  CA FF 0A 00  00 00 E4 A0  07 00 00 0A  ................
000001D0  CA FF 83 0A  CA FF EE A0  07 00 4A 42  0F 00 00 0A  ..........JB....
000001E0  CA FF 06 0A  CA FF 38 E3  16 00 78 CC  24 00 00 00  ......8...x.$...
000001F0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 55 AA  ..............U.
```

Figure 15 : Descripteur de partitions, la partition 4 (vert foncé) n'est pas utilisée.

Cette partie du MBR (Fig. 15) contient quatre entrées de 16 octets, chacune d'elles fournissant l'ensemble des informations sur une partition. Toutefois, la seule donnée importante pour la suite est la position du début de la partition (cadre rouge vif), se trouvant à l'offset 0x1C6 (dans le cas de la première partition).

Une fois la position de la partition trouvée, il est possible de se rendre à cette adresse afin de pouvoir exploiter celle-ci.

8.2 Structure

Avant de présenter le système de fichiers en lui-même, il est nécessaire de préciser à quoi correspond un cluster. Au sens de ce système de fichiers, c'est l'unité de base. Il correspond à un agglomérat de n secteurs. La taille est contenue dans la zone décrivant la partition.

Le système de fichiers FAT16 est structuré de la manière suivante :

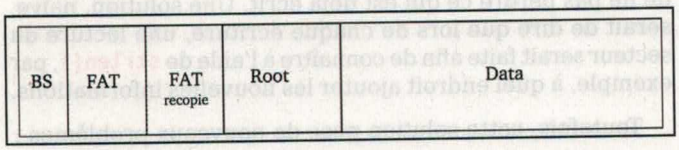


Figure 16 : Structure d'une partition FAT16

8.2.1 Boot Sector

Au début de la partition se trouve le *Boot Sector* (BS). Il a le même rôle au niveau de la partition que le MBR pour le support de stockage entier. Il contient l'ensemble des informations nécessaires à l'exploitation de la partition.

Entre autres, le BS contient la taille :

- de la partition ;
- des diverses zones ;
- d'un cluster ;
- etc.

Seul le BS possède une taille définie dans les spécifications, l'accès au reste des zones de la partition se fait grâce aux informations issues de cette zone.

8.2.2 FAT (File Allocation Table)

Dans la suite, afin d'éviter les confusions, le système de fichiers sera désigné par le mot « FAT16 », tandis que les listes chaînées le seront par « FAT ».

Le système de fichiers contient deux FAT. La seconde sert de sauvegarde et devrait (si tout s'est bien passé) être la copie conforme de la première.

Cette structure est une liste simplement chaînée de clusters. Le contenu du fichier étant stocké par paquets de n secteurs non consécutifs, la relation entre chacun d'eux se fait grâce à la FAT. Le cluster débutant les données d'un fichier est indiqué par l'entrée de fichier dans la zone RootDirSector.

Les premiers 16 bits de la FAT donnent le type du médium, les seconds donnent l'état de la partition. Le premier cluster exploitable de cette zone possède le numéro 2, le dernier dépend de la taille de la partition. Chaque index de clusters est codé sur 16 bits, la valeur d'un index pouvant être de trois types :

- égale à **0x0000**, pour signifier que le cluster n'est pas utilisé et peut donc être réservé ;
- égale à **0xFFFF**, pour signifier que d'une part, le cluster est déjà réservé mais qu'en plus, il correspond au cluster de fin de fichier ;
- avec une valeur comprise entre les deux précédentes pour donner l'index de son successeur dans la liste.

La figure17 présente un exemple simple, dans lequel il existe deux fichiers. Le contenu du premier :

- débute au cluster 2 (information obtenue depuis l'entrée du fichier) ;
- se continue au cluster 3 (valeurs du cluster 2) ;
- pour se finir au 9 (possède la valeur 0xffff).

Il en va de même pour le second fichier commençant au cluster 6.

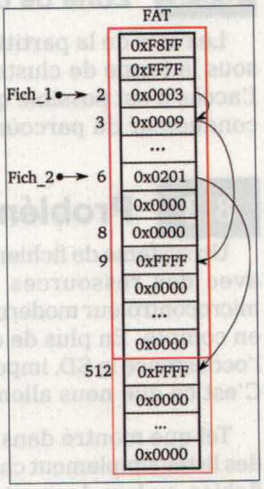


Figure 17 : Exemple de FAT

8.2.3 Répertoire racine

Le répertoire racine de la partition (RootDirSector) contient l'ensemble des fichiers et répertoires se trouvant sous la racine, chaque entrée de fichier fournissant l'ensemble des données relatives à celui-ci. Les plus importantes étant l'index du premier cluster des données ainsi que la taille du fichier.

```
0003E600   41 74 00 6F  00 74 00 6F  00 2E 00 0F  00 83 74 00   At.o.t.o......t.
0003E610   78 00 74 00  00 00 FF FF  FF FF 00 00  FF FF FF FF   x.t.............
0003E620   54 4F 54 4F  20 20 20 20  54 58 54 20  00 64 B8 5E   TOTO    TXT .d.^
0003E630   85 3B 85 3B  00 00 B8 5E  85 3B 02 00  00 00 00 00   .;.;...^.;......
0003E640   E5 74 00 65  00 6D 00 70  00 2E 00 0F  00 81 74 00   .t.e.m.p......t.
0003E650   78 00 74 00  00 00 FF FF  FF FF 00 00  FF FF FF FF   x.t.............
0003E660   E5 45 4D 50  20 20 20 20  54 58 54 20  00 64 B8 5E   .EMP    TXT .d.^
0003E670   85 3B 85 3B  00 00 B8 5E  85 3B 00 00  00 00 00 00   .;.;...^.;......
0003E680   42 2E 00 74  00 78 00 74  00 00 00 0F  00 8A FF FF   B..t.x.t........
0003E690   FF FF FF FF  FF FF FF FF  FF FF 00 00  FF FF FF FF   ................
0003E6A0   01 6E 00 6F  00 6D 00 5F  00 74 00 0F  00 8A 72 00   .n.o.m._.t....r.
0003E6B0   65 00 73 00  5F 00 6C 00  6F 00 00 00  6E 00 67 00   e.s._.l.o...n.g.
0003E6C0   4E 4F 4D 5F  54 52 7E 31  54 58 54 20  00 64 87 62   NOM_TR~1TXT .d.b
0003E6D0   85 3B 85 3B  00 00 87 62  85 3B 09 00  00 00 00 00   .;.;...b.;......
0003E6E0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
0003E6F0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
```

Figure 18 : Répertoire racine d'une partition FAT16

Pour chaque entrée de fichier, le premier octet a une valeur particulière. Lorsqu'il vaut :

- **0x00**, le parcours est fini. Il n'y a plus de fichiers ensuite (comme l.15 de 18).
- **0xE5**, le fichier est effacé (l.5 et 7).
- **0x4n**, pour le début du nom long, le n donne le nombre de lignes utilisées pour stocker le nom long (n appartient à [1;9]) (l.9),
- Dans le reste des cas, il correspond au premier caractère du nom du fichier (l.3 et l.13).

La recherche d'un fichier se fait donc en parcourant cette zone et en étudiant chaque entrée de fichier une à une.

Si l'on ne s'intéresse qu'au nom DOS (nom court), le parcours se fait de la façon suivante : au départ du parcours, on peut s'attendre soit à un fichier supprimé, soit à un nom long.

- Dans le premier cas, il suffit de passer 32 octets pour arriver sur l'entrée suivante.
- Dans le second cas, il faut passer n * 32 octets pour parvenir au nom DOS (format 8.3).

8.2.4 Zone de données

Le reste de la partition contient les données structurées sous la forme de clusters, eux-mêmes divisés en secteurs. L'accès n'est possible qu'en se servant de la FAT comme fil conducteur du parcours, tel que présenté précédemment.

8.3 Problématique

Un système de fichiers, bien que créé pour des ordinateurs avec des ressources quasi équivalentes à celles d'un microcontrôleur moderne, entraîne des contraintes à prendre en compte. En plus de celles-ci, le support de stockage, en l'occurrence la SD, impose également son lot de contraintes. C'est ce que nous allons voir maintenant.

Tel que montré dans la section précédente, les FAT sont des listes simplement chaînées. Ainsi, lors de l'ouverture d'un fichier, ou lors du montage de la partition, il est nécessaire de retrouver le dernier cluster des données du fichier et le premier cluster libre. Ceci se fait par des « sauts de puce », dans le cas du fichier, en passant d'une entrée de cluster à sa suivante. Dans le cas de la recherche d'un cluster libre, toutes les entrées contenues dans un secteur sont balayées. Dans les deux cas, dès lors que le cluster suivant se trouve dans un autre secteur, il devient nécessaire de faire une nouvelle lecture sur le support physique. Dans le cas d'une application se réveillant périodiquement pour faire un stockage, ceci peut donc entraîner la lecture d'un très gros volume de données et donc augmenter le temps nécessaire à l'initialisation, ainsi que la consommation d'énergie. Une solution serait de stocker ces informations quelque part, tel que dans un fichier de configuration. Seulement cette solution ne serait valable que dans le cas d'un fichier, car imposant d'avoir déjà la partition initialisée, et par ailleurs, nécessiterait un parcours additionnel pour l'ouverture de ce fichier. Une autre solution serait de stocker ces informations dans une zone non utilisée du BS, mais ceci peut compromettre d'autres informations.

La solution mise en œuvre consiste à mettre en veille et à réveiller le descripteur de fichier et la partition. La seule différence, par rapport à un arrêt et un démarrage, réside dans la conservation des numéros de clusters en mémoire. Les gains de cette solution ne sont, certes, que partiels, limités au cas d'un réveil sur une application périodique, car lors du démarrage de l'application, il est nécessaire de rechercher les clusters. Toutefois, et c'est également le cas pour la solution basée sur la lecture d'informations stockées quelque part sur la carte, il est préférable de considérer que la seule source d'informations valable est issue de la FAT16.

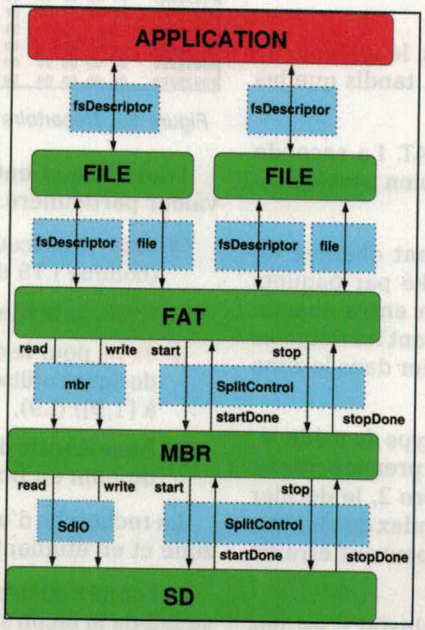


Figure 19 : Structure globale de l'implémentation

Le second problème concerne le stockage des données acquises. En effet, la SD imposant l'écriture de 512 octets, il faut pouvoir concaténer les nouvelles données.

Dans le cas de l'écriture permanente de ce même volume d'informations, ceci ne pose pas de problème, mais si chaque écriture concerne un volume plus faible, il devient nécessaire de ne pas perdre ce qui est déjà écrit. Une solution, naïve, serait de dire que lors de chaque écriture, une lecture du secteur serait faite afin de connaître à l'aide de `strlen()`, par exemple, à quel endroit ajouter les nouvelles informations.

Toutefois, cette solution pose de nouveaux problèmes :

- Dans le cas de l'écriture de 512 octets, la phase de lecture est inutile voire même néfaste sur l'autonomie.
- Pour pouvoir connaître le volume de données déjà stockées, il devient nécessaire d'assurer que de précédentes données ne viendraient pas parasiter le comptage, en d'autres termes, que le secteur soit « nettoyé » avant son utilisation. Pour cela, lors de la réservation d'un cluster, il faudrait remettre à zéro l'ensemble des secteurs contenus dans celui-ci, entraînant par la même occasion la multiplication par deux du volume écrit.
- Pour finir avec les inconvénients de cette solution, dans le cas, par exemple, de l'utilisation de certains GPS dont les informations sont binaires, le comptage risque d'être faussé par la présence d'octets ayant pour valeur `0x00`.

Ce problème a été résolu grâce à une variable se trouvant au niveau du descripteur de fichier : celle-ci, initialisée par défaut par `bytesUsed = fileLength & 0x1FF;`, permet de connaître le nombre d'octets utilisés dans le dernier secteur. Elle permet donc de savoir :

- s'il est possible d'écrire directement (si égale à 0) ;
- de passer au secteur et/ou cluster suivant (si égale à 512) ;
- ou s'il faut faire une lecture et ajouter les nouvelles données à la suite de celles existantes. Un simple `memcpy` utilisant cette variable permet cet ajout, sans faire l'hypothèse d'un contenu « propre ».

Maintenant que les difficultés ont été présentées, nous allons voir le fonctionnement des modules ainsi que les points les plus importants en termes d'algorithmes.

8.4 Implémentation

L'ensemble des modules composant l'implémentation de la FAT16 se présente ainsi :

Toute l'implémentation ne sera pas présentée, les documents **[14]** et **[13]** présentent la globalité de ce système de fichiers. D'autre part, les sources du pilote sont disponibles à l'adresse http://www.trabucayre.com/tinyos/fat.html.

L'implémentation réalisée ne permet que l'écriture de données. Il n'est pas possible depuis une application de lire le contenu d'un fichier. Ceci s'explique par le fait qu'un stockage en mémoire non volatile peut avoir plusieurs usages en ce qui concerne la lecture :

- le stockage temporaire des informations en vue d'une transmission radio (par exemple) ;
- le stockage de paramètres de configuration pour le nœud.

Dans ces deux cas, il nous a semblé qu'un système de stockage avec lecture n'est pas adapté :

- Dans le premier cas, la solution du système de stockage formaté est faite pour éviter l'aspect transmission et augmenterait la consommation globale.
- La seconde raison vient du fait que TinyOS fournisse une solution utilisant les puces de flash de la plupart des cartes commerciales. Cette solution est mieux adaptée pour le stockage de configuration et de données temporaires en attente de transmission.

La suite de la présentation se fera de bas en haut sur le schéma, en commençant par le MBR.

8.4.1 MBR

La dernière couche avant la SD est un module gérant le MBR du support de stockage, son rôle se borne à :

- transmettre la commande de démarrage et d'arrêt au support de stockage ;
- trouver la position du début de la partition lors de son initialisation :

```
[...]
if (call SdIO.read(0,buf,512) == SUCCESS){
        debPartition =  (*(uint32_t *)&buf[pos+8])<<9;
        mbrState = MBR_IDLE;
        error = SUCCESS;
}
[...]
```

- ajouter cet offset lors de l'écriture (ou de la lecture s'il y a lieu) des données.

```
command error_t mbr.read(uint32_t offset, uint8_t *buffer) {
        [...]
        error = call SdIO.read(debPartition+offset,buffer,512);
        [...]
}
command error_t mbr.write(uint32_t offset, uint8_t *buffer) {
        [...]
        error=call SdIO.write(debPartition+offset,buffer,512);
        [...]
}
```

8.4.2 FAT16

Ensuite vient le module FAT, qui réalise les plus gros traitements : lors du démarrage du système de fichiers, il commence par lancer l'initialisation du MBR et analyse le BS afin d'extraire l'ensemble des informations nécessaires.

```
[...]
if (call mbr.read(0,buf) == SUCCESS) {
        BytsPerSec = (buf[12]<<8)+buf[11];
        SecPerClus=buf[13];
        RsvdSecCnt = (buf[15]<<8)+buf[14];
        NumFATs=buf[16];
        RootEntCnt = (buf[18]<<8)+buf[17];
        FATSz = (buf[23]<<8)+buf[22];
        FirstFatByts = RsvdSecCnt*BytsPerSec;
        // Repertoire racine
        RootDirByts = (RsvdSecCnt + NumFATs*FATSz)*BytsPerSec;
        // position premier secteur de donnees
        FirstDataSector = RootDirByts +
                fatAlignSup(RootEntCnt,BytsPerSec)*BytsPerSec;
        CountOfClusters = FATSz*((BytsPerSec/2)-2);
        if (fatState != FAT_SUSPEND)
                lastFreeCluster = 3;// Commence au début
[...]
```

Il se charge de la réservation des clusters : ceci se passe exclusivement au niveau de la FAT du système de fichiers. La première étape consiste à obtenir la position, en termes d'offset et de secteur, de la dernière entrée de cluster précédemment réservée (ou de la première disponible). L'offset, d'une part, donne la position par rapport au début du secteur, en cours d'utilisation, dans la FAT et d'autre part, permet de déterminer si toutes les entrées de cluster contenues dans ce secteur ont été parcourues.

```
[...]
getOffsetAndSector(lastFreeCluster,&fatSec,&off);
if (off < 512) {
        currentFatSec = (512*(fatSec-1))+FirstFatByts;
        if (call mbr.read(currentFatSec, buf) == FAIL)
                goto end;
}
```

Ensuite, le secteur est parcouru jusqu'à trouver un cluster libre. Lorsque le secteur en cours a été entièrement parcouru, la lecture du secteur suivant est faite.

```
do {
        if (off >= 512) { // Si on dépasse le secteur
                fatSec++; // passage secteur suivant
                currentFatSec = (512*(fatSec-1))+FirstFatByts;
                off = 0;  // raz de l'offset
                // Lecture d'un nouveau secteur de donnée
                if (call mbr.read(currentFatSec, buf) == FAIL)
                        goto end;
        }
        // Secteur trouvé maintenant faut faire propre
        if ( (*(uint16_t*)&buf[off]) == 0x00){
                error = SUCCESS;
                break;
        }
        lastFreeCluster++; // Passage au cluster suivant
        off+=2; // Prochaine position à lire
} while(lastFreeCluster < CountOfClusters);
[...]
```

La boucle se fait tant qu'un cluster libre n'a pas été trouvé ou que la FAT n'a pas été complètement parcourue.

Une fois un cluster trouvé, il est marqué comme utilisé en tant que fin de fichier.

```
(*(uint16_t *)&buf[off])=0xffff;
[...]
```

Puis les deux FAT sont remises à jour afin que l'ancien dernier cluster prenne comme valeur l'index du nouveau et que le nouveau soit noté comme utilisé en tant que cluster final de la chaîne des données du fichier. Pour la mise à jour, deux cas de figure sont possibles :

- Les deux clusters se trouvent dans le même secteur de la FAT. Dans ce cas, une seule écriture par FAT est nécessaire.

```
if (*cluster != 0) // au cas ou pas de precedent a mettre a jour
        (*(uint16_t *)&(buf[offOr]))=lastFreeCluster;
if (call mbr.write(currentFatSec, buf) == FAIL ||
         call mbr.write(currentFatSec+(FATSz*512),buf)==FAIL)
        goto end;
}
[...]
```

- Les deux clusters sont dans deux secteurs différents. Dans ce cas, il sera nécessaire de les remettre à jour en deux écritures successives.

```
if (fatSec != secOr) {
        if (call mbr.write(currentFatSec, buf)== FAIL ||
                call mbr.write(currentFatSec+(FATSz*512),
                buf)== FAIL)
                goto end;
        if (*cluster != 0) {
                tmp = (512*(secOr-1))+FirstFatByts;
                if (call mbr.read(tmp, buf)== FAIL) goto end;
                (*(uint16_t *)&(buf[offOr]))=lastFreeCluster;
                if (call mbr.write(tmp, buf) == FAIL ||
                        callmbr.write(tmp+(FATSz*512),buf)==FAIL)
                        goto end;
        }
```

Une fois les FAT mises à jour, les modifications sont répercutées sur les variables.

```
*cluster = lastFreeCluster;
lastFreeCluster++; // Passe au suivant vu que le courant est pris
```

La troisième et dernière tâche importante se passe lors de l'écriture de données : si le secteur qui va être utilisé dépasse le nombre de secteurs du cluster en cours, une réservation de cluster est faite.

```
[...]
if (*secteur >= SecPerClus || *cluster == 0) {
        //remise à jour de la valeur de cluster
        if (fatReserveCluster(buf, cluster) == FAIL)
                goto end;
        *bytesUsed = 0; // Nouveau cluster donc rien deja ecrit
        *secteur = 0;
}
```

Ensuite, la position absolue (depuis le début de la SD) du secteur courant (en octet) est calculée.

```
// calcul de la position
sector = computeSecFromCluster(*cluster)+((*secteur)*512);
memset(buf,'\0',512);
```

Si le secteur en cours d'utilisation n'est pas vide, son contenu est récupéré afin de faire une concaténation.

```
/* lecture de cette position seulement si deja du contenu */
if (*bytesUsed != 0) {
        if (call mbr.read(sector, buf) == FAIL)
                goto end;
}
```

Il évalue ensuite la taille des données déjà présentes, plus la taille des données à écrire, afin de savoir si elles dépassent la taille du secteur.

```
if (*bytesUsed + s > 512){
        (*secteur)++; // La suite dans le prochain secteur
        size = 512 - *bytesUsed;
} else {
        if (*bytesUsed + s == 512)      (*secteur)++;
        size = s;
}
```

Et pour finir, remplit le secteur, puis fait une mise à jour du numéro du secteur ainsi que du nombre d'octets écrits.

```
memcpy(buf+*bytesUsed,c,size);
if (call mbr.write(sector,buf) == FAIL)
        goto end;
s-=size;
*bytesUsed = (*bytesUsed+size) & 0x1FF;
c += size;
[...]
```

Et ceci tant que l'ensemble des informations ne sont pas complètement écrites.

8.4.3 file

Son principe est assez similaire au descripteur de fichier dans un système d'exploitation. Il gère tous les aspects liés spécifiquement au fichier en lui-même.

A l'initialisation, l'entrée pour le fichier concerné est recherchée :

```
[...]
// recup du premier secteur du root
if (call fat.fatReadRoot(0,buf) == FAIL)
        return ret;
do {
        // parcours du secteur
        c = buf+offset;
        if (c[0] == 0x00) { // Fin de fichier
                break; // Rien a trouver
        // un nom de fichier long, a skipper
        } else if ((uint8_t)c[0] == 229 || c[11] == 15) {
        } else if (c[11] == 0x20) { // Trouve un fichier
                strncpy(fileName,c,7);
                fileName[8]='\0';
                if (!strncmp(fileName,name,strlen(name))) {
                        ret = SUCCESS;
                        break;
                }
        }
        // passage a l'entree suivante
        offset += 32;
        // si depassement du secteur faut lire le suivant
        if (offset >= 512) {
                sector++;
                offset -= 512;
                call fat.fatReadRoot(sector,buf);
        }
}while(c[0] != 0x00);
// renvoi des valeurs
*off = offset;
*sect = sector;
[...]
```

Une fois celui-ci trouvé, l'ensemble des variables est rempli. Dans le cas où le fichier serait vide (numéro de cluster égal à zéro), il est fait une demande de réservation. Si le fichier a été mis en veille (commande **suspend()**), la recherche du cluster de fin de fichier ne sera pas réalisée car la variable contient déjà cette information.

```
/* Enregistrement du nom de fichier */
c = (buf+offset);
// Position premier cluster du contenu
dataLocation = c[26];
// Longueur du fichier
fileLength = (*(uint32_t *)&(c+sec)[28]);
// Position dans le root dir sector de l'entrée
rootSector = sec;
rootOffset = offset;
// Cluster de fin du fichier
// Si le fichier est vide
if (dataLocation < 2) {
        // Réservation d'un cluster d'emblé
        if (call fat.reserveCluster(&dataLocation,buf) == FAIL){
                return FAIL;
        }

        lastCluster = dataLocation;
        lastSecteur = 0;
        bytesUsed = 0;
        call fat.fatReadRoot(rootSector,buf);
        (*(uint16_t *)&(buf+rootOffset)[26]) = dataLocation;
        call fat.fatWriteRoot(rootSector,buf);
} else {
        if (fileState == FILE_NOINIT) {
            if ((lastCluster = call fat.fatLastCluster(dataLocation,buf))

== -1)
                        return FAIL;
            if ((lastSecteur = call fat.fatLastSecteur(lastCluster,buf))

== -1)
                        return FAIL;
        }
        bytesUsed = fileLength & 0x1FF;
}
```

Lors d'une demande d'écriture, le tableau contenant les données est passé au système de fichiers, ainsi que le nombre d'octets déjà écrits. À l'issue de l'écriture des données, la taille du fichier est mise à jour. Ce comportement, bien que nécessitant une écriture de plus, apporte une sûreté quant aux données écrites. En effet, si une coupure d'alimentation avait lieu, dans le cas contraire, il ne serait pas possible de récupérer les données facilement.

8.5 Exemple d'utilisation

Pour montrer comment utiliser cette implémentation de la FAT16, nous allons présenter deux exemples. Le premier se contente de lancer le système de fichiers, d'ouvrir le fichier **toto.txt**, dans lequel il écrit « hello World », puis finit en fermant le fichier et le système de fichiers. Le second a pour but de montrer l'utilisation des méthodes de mise en sommeil et de réveil du système de fichiers en lisant et stockant la température, à intervalles réguliers.

8.5.1 Utilisation de la FAT16

```
configuration fatTestAppC {}
implementation {
        components fatTestC as App, MainC, fatC;
        components new TimerMilliC() as Timer0;
        components new fileC("toto.txt");
        App.Boot -> MainC.Boot;
        App.Timer0 -> Timer0;
        App.fatControl -> fatC;
        App.file -> fileC;
        App.fileControl -> fileC.fileControl;
        fileC.fat -> fatC.fat;
}
```

Listing 9 : Fichier de configuration de l'application

Le premier fichier (Fig. 9) câble l'ensemble des modules, avec dans le cas du fichier la spécification du nom de celui-ci.

```
module fatTestC {
        uses {
                interface Boot;
                interface Timer<TMilli> as Timer0;
                interface SplitControl as fatControl;
                interface SplitControl as fileControl;
                interface fat;
                interface file;
        }
}
implementation {
        event void Boot.booted()  {
                call Timer0.startOneShot(500);
        }
        event void Timer0.fired() {
                call fatControl.start();
        }
        event void fatControl.startDone(error_t err){
                if (err == SUCCESS) call fileControl.start();
        }
        event void fileControl.startDone(error_t err) {
                if (err == SUCCESS)
                        call file.write("hello world",11)
        }
        event void file.writeDone(error_t err) {
                if (err == SUCCESS) call fileControl.stop();
        }
        event void fileControl.stopDone(error_t err) {
        if (err == SUCCESS) call fatControl.stop();
        }
        event void fatControl.stopDone(error_t err){}
        [...]
}
```

Listing 10 : Fichier de l'application

Le second (listing 10) :

- monte la partition lorsque le *timer* arrive à expiration (l.15-17) ;
- si la partition est correctement montée, il lance l'ouverture du fichier (l.18-20) ;
- une fois le fichier ouvert, il fait une écriture de « hello World » dedans (l.21-24) ;
- une fois le contenu écrit, ferme dans l'ordre inverse d'ouverture le fichier et la partition (l.25-31).

8.5.2 Stockage de la température et mise en veille

```
configuration storeTempAppC {}
implementation {
        components storeTempC as App, LedsC, MainC;
        App.Boot -> MainC.Boot;
        App.Leds -> LedsC;
        components new TimerMilliC() as Timer0;
        App.Timer0 -> Timer0;
        components fatC;
        App.fatDescriptor -> fatC;
        components new fileC("temp.txt") as fileADC;
        App.fileADC -> fileADC;
        App.fileADCDescriptor -> fileADC.fileDescriptor;
        fileADC.fat -> fatC.fat;
        components new DemoSensorC() as Sensor;
        App.readADC -> Sensor;
}
```

Figure 20 : Fichier de configuration de l'application d'acquisition de température

Le premier fichier (Fig. 20) contenant la configuration de l'application est globalement le même que celui de l'exemple précédent, hormis l'utilisation de **DemoSensorC**, utilisé pour la température et le nom du fichier qui sera utilisé.

```
#include "Timer.h"
#define TIMER_SLEEP 3600000
#define SHOW_ERROR do { \
        call Leds.led1Off(); \
        call Leds.led0On(); } while(0)

module storeTempC {
        uses {
                interface Leds;
                interface Boot;
        interface Timer<TMilli> as Timer0;
                interface Read<uint16_t> as readADC;
        interface fat;
                interface fsDescriptor as fatDescriptor;
                interface file as fileADC;
                interface fsDescriptor as fileADCDescriptor;
        }
}
```

La vue externe est elle aussi globalement la même, avec seulement l'ajout de l'interface permettant l'exploitation de l'ADC12 du MSP430.

La partie implémentation peut être divisée en quatre grands blocs fonctionnels.

```
implementation {
        enum {
                APP_NOINIT,
                APP_STOP,
                APP_SLEEP,
                APP_ADC
        };
        uint8_t appState = APP_NOINIT;
        uint8_t *tampon=NULL;

        event void Boot.booted() {
                tampon          = (uint8_t *) malloc(8*sizeof(uint8_t *));
                appState        = APP_NOINIT;
                call Timer0.startPeriodic(TIMER_SLEEP);
        }
        event void Timer0.fired() {
                error_t error = FAIL;
                call Leds.led1Off();
                call Leds.led0Off();
                if (appState == APP_NOINIT || appState == APP_STOP)
                        error = call fatDescriptor.open();
                else if (appState == APP_SLEEP)
                        error = call fatDescriptor.resume();
                if (error == FAIL)
                        SHOW_ERROR;
        }
```

La première concerne la définition de variables nécessaires pour conserver l'état de l'application au fil du temps et d'un tableau nécessaire à l'écriture sur la carte. Nous définissons plusieurs états (l.2-7) et une variable pour les stocker, afin que lors de l'expiration du timer (l.16-26), l'application soit en mesure d'ouvrir le système de fichiers (appel à **call fatDescriptor.open()**) ou de le réveiller (appel de **call fatDescriptor.resume()**).

Le reste concerne l'initialisation du timer et la gestion de l'expiration de celui-ci.

```
        event void fatDescriptor.openDone(error_t error){
                if (error == SUCCESS)   error = call fileADCDescriptor.open();
                if (error == FAIL)              SHOW_ERROR;
        }
        event void fileADCDescriptor.openDone(error_t error) {
                if (error == SUCCESS){
                        atomic { appState = APP_ADC;}
                        error = call readADC.read();
                }
                if (error == FAIL) {
                        call fatDescriptor.close();
                        SHOW_ERROR;
                }
        }
        event void fatDescriptor.resumeDone(error_t error){
                if (error == SUCCESS)  error = call fileADCDescriptor.resume();
                if (error == FAIL)              SHOW_ERROR;
        }
        event void fileADCDescriptor.resumeDone(error_t error) {
                if (error == SUCCESS){
                        atomic { appState = APP_ADC;}
                        error = call readADC.read();
                }
                if (error == FAIL) {
                        call fileADCDescriptor.close();
                        SHOW_ERROR;
                }
        }
```

Le second bloc concerne les aspects liés au démarrage/réveil du système de fichiers. Comme nous pouvons le voir, il n'y a pas une grosse différence au niveau de l'application entre démarrage et réveil. Lors de la réussite de l'opération au niveau du système de fichiers, le fichier lui-même est relancé et à la fin de l'opération, une acquisition est faite.

```
        event void fileADCDescriptor.closeDone(error_t error) {
                if (error == SUCCESS)   error = call fatDescriptor.close();
                if (error==FAIL)                SHOW_ERROR;
        }
        event void fatDescriptor.closeDone(error_t error){
                if (error == FAIL)      SHOW_ERROR;
                else                            atomic {appState = APP_STOP;}
        }
        event void fileADCDescriptor.suspendDone(error_t error) {
                if (error == SUCCESS)   error = call fatDescriptor.suspend();
                if (error == FAIL)              SHOW_ERROR;
        }
        event void fatDescriptor.suspendDone(error_t error){
                call Leds.led1Off();
                if (error == FAIL)              SHOW_ERROR;
                else                            atomic { appState = APP_SLEEP;}
        }
}
```

Le troisième bloc est le pendant du précédent et concerne la gestion de l'arrêt/mise en veille. Les opérations étant faites dans l'ordre contraire du démarrage.

```
        event void readADC.readDone(error_t result, uint16_t data) {
                uint16_t inter, i, z;
                float val = (((data/4096.0)*1.5)-0.986)/0.00355;
                memset(tampon,'\0',8*sizeof(uint8_t));
                for(i=0,z=100;i<3;i++,z/=10){
                        inter = val / z;
                        tampon[i] = inter+'0';
                        val -= inter*z;
                }
                tampon[3] = '\n';
                appState = APP_SLEEP;
                if (call fileADC.write(tampon,4) == FAIL)
                        SHOW_ERROR;
        }
        event void fileADC.writeDone(error_t error) {
                if (error == SUCCESS)
                        error = call fileADCDescriptor.suspend();
                if (error == FAIL){
                        call fileADCDescriptor.close();
                        SHOW_ERROR;
                }
        }
```

Le dernier correspond au traitement à proprement parler. Bien que ce soit la partie la plus courte de l'application, c'est aussi la partie qui contient tout le fonctionnement de celle-ci.

`readADC.readDone(...)` est exécutée lorsque le microcontrôleur a fini la lecture de la température. Après une conversion en chaîne de caractères, une demande d'écriture est faite. Quand cette dernière est finie, le système de fichiers est mis en sommeil jusqu'à la prochaine expiration du timer périodique.

8.6 Test de vitesse et de portabilité

Pour connaître la vitesse d'écriture, une application a été réalisée. Celle-ci est équivalente à celle pour la mesure du débit de la SD en rawWrite. Le test a duré 5 min 41 s, présentant une vitesse approximative de 3,2 kB/s. Comparé à la vitesse de rawWrite, trois fois plus élevée, le résultat, compte-tenu du nombre d'opérations réalisées, est cohérent.

Cette vitesse est largement suffisante pour une application pratique de stockage de trames GPS - incluant l'utilisation d'un récepteur fournissant l'information de phase tel que le Thalès AC12 - dont le débit mesuré peut être au maximum de 3200 b/s (ou 0,4 kB/s).

L'ensemble du travail de développement des pilotes a été réalisé sur notre plate-forme expérimentale. Toutefois, afin de pouvoir valider la bonne intégration de l'ensemble dans TinyOS, nous les avons également testés sur des plates-formes commerciales, nativement supportées par TinyOS.

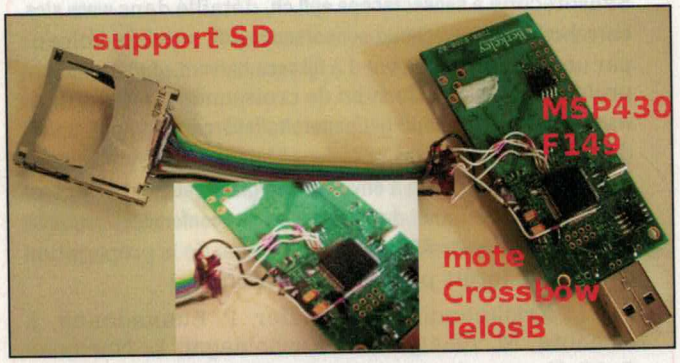


Figure 21 : Carte TelosB - commercialisée par Crossbow - équipée d'une carte SD

La première, la TelosB (Fig. 21), est également basée sur un MSP430. L'utilisation de la SD ainsi que de la FAT16 s'est bornée à copier les fichiers de configuration de la plate-forme et à reconfigurer le SPI pour faire usage du quartz 32 kHz (la TelosB ne contient pas de quartz haute fréquence). Comme l'architecture du microcontrôleur MSP430F1611 est la même que celle utilisée sur notre carte de prototype, la liaison de la carte SD s'effectue en connectant les mêmes broches que dans notre exemple : Chip Select sur P3.0 (broche 28), MOSI sur P3.1 (broche 30), MISO sur P3.2 (broche 30) et l'horloge sur P3.3 (broche 31).

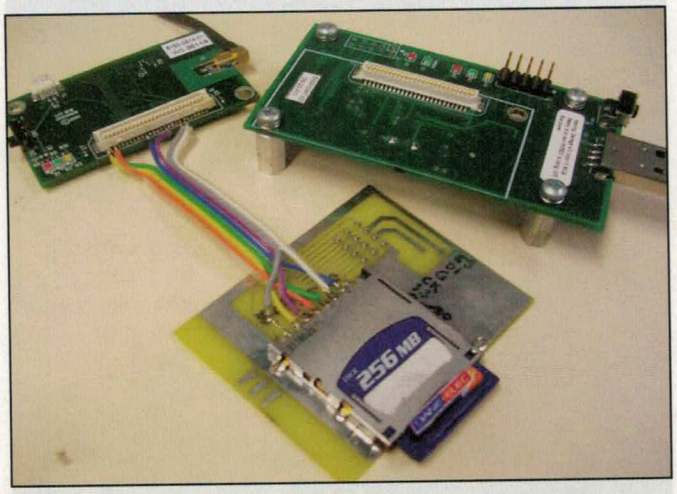


Figure 22 : Carte MicaZ - commercialisée par Crossbow - équipée d'une carte SD

La seconde plate-forme est une MicaZ, celle-ci est sur une base d'un ATMEGA128 **[16]**. La mise en œuvre des pilotes sur celle-ci n'a pas pris plus d'une trentaine de minutes, après identification du fonctionnement du SPI sur cette carte. En effet, TinyOS utilise une version logicielle et non matérielle de ce protocole. Sur cette architecture, la carte SD est reliée aux broches suivantes du connecteur 51 broches de la carte MicaZ :

- Chip Select est connecté à la broche LED1 (broche 10, PA2) ;
- MISO est connecté à la broche USART1 RXD (broche 19, PD2) ;
- MOSI est connecté à la broche USART1 TXD (broche 20, PD3) ;
- finalement, l'horloge (*Clock*) est fournie par USART CLK (broche 15, PD5).

CONCLUSION

Nous avons proposé quelques exemples de développements autour de TinyOS-2.x sur plate-forme MSP430, en partant des cas les plus simples des entrées/sorties numériques, pour ensuite exploiter les ports de communication série asynchrone et synchrone, et conclure sur l'implémentation d'un système de stockage de fichiers sur support formaté compatible avec la majorité des systèmes d'exploitation sur PC.

Le premier résultat concerne les pilotes qui ont été réalisés : afin d'évaluer la validité à la fois du pilote GPS et du système de fichiers, la plate-forme a été exploitée en pratique avec une alimentation sur accumulateurs pour l'acquisition de trames GPS. Une autonomie de plus de 25 h a ainsi pu être validée lors de l'acquisition de traces GPS de plusieurs MB sans corruption de la carte mémoire.

Après récupération du fichier sur un ordinateur, la trace GPS est traitée au moyen de GpsQt 3 pour insertion sur un fond de carte Google Maps (Fig. 23, d'autres itinéraires sont disponibles sur http://www.trabucayre.com/gps).

Figure 23 : Tracé d'un trajet entre la région de Brest et le nord de Besançon (en rouge), sur un fond de carte Google Maps : 950 km pendant lesquels le circuit a fonctionné sur 4 accumulateurs NiMH pendant 11 h. Le résultat est un fichier de 142411 lignes pour une taille de 9,2 MB.

À temps de développement équivalent qu'en C ou en assembleur, pour une application proposant les mêmes fonctionnalités, le principal intérêt de TinyOS tient en la possibilité de réutiliser le code sans avoir à reprendre l'ensemble de l'implémentation de la communication avec la carte SD et le format FAT.

Par ailleurs, nous avons constaté que l'ajout de pilotes pour des périphériques non supportés par défaut (par exemple, l'écran LCD compatible HD44780) est simplifié par la ressemblance syntaxique du nesC et du C : une implémentation du protocole existante en C est rapidement adaptée aux contraintes de TinyOS. Ceci permet également de réaliser une première implémentation des spécifications sur un ordinateur afin de valider le code avant de le convertir en un module TinyOS, comme ce fut le cas avec la FAT.

Les routines offertes par TinyOS donnent la possibilité au programmeur de se concentrer sur une tâche bien ciblée, sans avoir à reprendre ou réadapter un autre code de communication avec le matériel. Il est, par exemple, trivial de réaliser une application utilisant une communication RS232 car le système offre d'emblée toutes les bibliothèques nécessaires à son exploitation.

TinyOS-2.x manque encore d'une certaine maturité. Il est, par exemple, étonnant de se voir contraint à un nombre prédéfini de LED, nécessitant dans certains cas de surcharger le module officiel. Le support du stockage n'est, contrairement au reste du système, pas indépendant du composant physique. Cela force donc à réimplémenter toutes les spécifications si le composant (par exemple, la carte SD) n'est pas supporté d'origine. Nous avons néanmoins constaté que la hiérarchie des pilotes et la possibilité de réutilisation de méthodes existantes pour l'accès au matériel rendent ce travail moins fastidieux qu'en développant du code de bas niveau sans exploiter les fonctionnalités de l'environnement exécutif.

Références

- **[1]** T. Liu, C.M Sadler, P. Zhang & M. Martonosi, *Implementing software on resource-constrained mobile sensors: experiences with impala and zebranet*, Proceedings de MobiSYS'04 (6-9 Juin 2004), disponible à www.cs.princeton.edu/~tliu/p206-liu.pdf
- **[2]** le projet TurtleNet est décrit à prisms.cs.umass.edu/dome/turtlenet
- **[3]** le projet Argo est décrit à www.argo.ucsd.edu, le projet COMMON-Sense Net est à commonsense.epfl.ch, le projet Sensorscope à senseorscope.epfl.ch, détaillé dans www.sics.se/realwsn05/slides/schmid-sensorscope.pdf, suivi de volcans par un groupe de Harvard à fiji.eecs.harvard.edu/Volcano, un projet de Berkeley de suivi de croissance d'arbres dans www.eecs.berkeley.edu/~get/papers/tolle05redwoods.pdf, suivi de glaciers à www.sics.se/realwsn05/papers/martinez05glacial.pdf (projet Glacsweb à envisens.org/glacsweb.htm) et projet Seamonsterak décrit dans esto.nasa.gov/conferences/estc2008/presentations/HeavnerA9P3.pdf, observation de la propagation de feux de forêts par firebug.sourceforge.net
- **[4]** L. Krishnamurthy, R. Adler, P. Buonadonna, J. Chhabra, M. Flanigan, N. Kushalnagar, L. Nachman & M. Yarvis, *Design and deployment of industrial sensor networks: experiences from a semiconductor plant and the north sea, Proc. of the 3rd international conference on embedded networked sensor systems*, pp.64-75, (2005)
- **[5]** Référence du langage nesC : http://nescc.sourceforge.net/papers/nesc-ref.pdf
- **[6]** la programmation sur TinyOS 2.x : http://www.tinyos.net/tinyos-2.x/doc/pdf/tinyos-programming.pdf
- **[7]** *Tutorial TinyOS 1.0* concernant la création d'une plate-forme : http://docs.tinyos.net/index.php/Platforms
- **[8]** *MSP430x1xx Family User's Guide* (2006) : focus.ti.com/lit/ug/slau049f/slau049f.pdf

- **[9]** J.-M. Friedt & S. Guinot, « Stockage de masse non volatil : un block device pour multimediacard », *GNU/Linux Magazine France* Hors-série n°25 (Avril/Mai 2006)
- **[10]** J.-M. Friedt & É. Carry, « Enregistrement de trames GPS - développement sur microcontrôleur 8051/8052 sous GNU/Linux », *GNU/Linux Magazine France* n°81 (Février 2006)
- **[11]** Présentation du protocole de communication des SD et MMC en SPI à http://www.retroleum.co.uk/mmc_cards.html
- **[12]** A. Bourgeois, « Croisière au cœur d'un OS : système de fichiers FAT », *GNU/Linux Magazine France* n°98 (Octobre 2007)
- **[13]** Explication de la structure d'un support de stockage : http://www.beginningtoseethelight.org/fat16/
- **[14]** Spécifications des partitions FAT : download.microsoft.com/download/1/6/1/161ba512-40e2-4cc9-843a-923143f3456c/fatgen103.doc
- **[15]** M. Tischer, *La bible du PC - programmation système*, 6ème édition, MicroApplications (1996)
- **[16]** D. Bodor, « Découverte du microcontrôleur AVR », *GNU/Linux Magazine France* Hors-série n°23 (2008)

Notes

[1] Nous n'avons pas expérimenté la programmation par communication série asynchrone au moyen du BSL décrit dans la note d'application SLAA089B de Texas Instruments.

[2] http://home.iae.nl/users/pouweha/lcd/lcd.html

Auteur : Gwenhaël Goavec-Merou

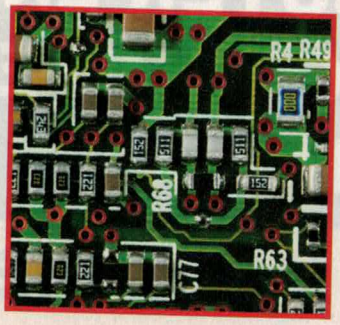
Gwenhaël Goavec-Merou a obtenu son master d'informatique de l'Université de Franche-Comté et travaille actuellement dans l'équipe temps-fréquence de l'Institut FEMTO-ST sur l'implémentation d'algorithmes de calculs dans les matrices de portes logiques reconfigurables (FPGA). Son projet de master était dédié au stockage de masse sur des nœuds de réseaux de capteurs faible consommation et la programmation sous TinyOS, activité qu'il a poursuivie durant l'année passée pour obtenir un résultat exploitable par la communauté des utilisateurs TinyOS.

Il est propriétaire de petits systèmes, type «chat», faible consommation (sieste), configurés, sur timer, pour un réveil périodique (8h, 12h, 19h) en vue d'acquisition (croquettes).

Auteur : Jean-Michel Friedt

Jean-Michel Friedt est ingénieur, hébergé par l'équipe temps-fréquence de FEMTO-ST et membre de l'association pour la diffusion de la culture scientifique et technique Projet Aurore. Son intérêt pour les capteurs actifs à faible consommation est principalement dû aux applications de mesures en environnements sévères ou sur de longues durées, fastidieuses voire impossibles sans ces outils.

Le VHDL pour les débutants



Auteur

- Laura Bécognée

« Ils ne savaient pas que c'était impossible, alors ils l'ont fait. » -- Mark Twain

Cet article n'a pas pour but d'être une description complète de ce qu'est le VHDL, mais simplement une initiation, sous forme d'exemple. Il s'adresse aux personnes qui, comme moi, ont décidé (pour une fois) de découvrir un langage avant de se renseigner sur ses difficultés inhérentes. Parce qu'il est parfois essentiel d'avancer sans préjugés. Il utilise la syntaxe de VHDL'87, qui suffit à la très grande majorité des cas. Un prochain article de Yann Guidon, à paraître dans GLMF 126, abordera quant à lui certaines techniques introduites par VHDL'93.

1 DÉFINITION GÉNÉRALE

Le VHDL est un langage de programmation permettant la description, la simulation et la synthèse de circuits électroniques. Son sigle signifie d'ailleurs *VHSIC Hardware Description Language*, où VHSIC signifie *Very High Speed Integrated Circuits*. (VHSIC comme VHDL sont à l'origine des initiatives du DARPA, l'agence technologique de l'armée américaine).

- La description est utile au niveau humain pour comprendre le fonctionnement interne d'un circuit électronique.
- La simulation permet, via un logiciel, de tester un circuit électronique. Ses performances, sa fiabilité, sa compatibilité avec d'autres circuits électroniques peuvent être vérifiées, ce qui permet de réduire considérablement le nombre de prototypes nécessaires et le risque de fabriquer des circuits verminés.
- Enfin, la synthèse permet d'implanter un circuit électronique complexe dans une puce composée de centaines voire de milliers de blocs logiques. Cette étape transforme la description de haut niveau (lisible par les humains) en fichiers de description de chaque fonction logique, avec un format et une structure propre à la technologie matérielle utilisée (ou cible).

On distingue principalement deux catégories de puces :

- Les FPGA (*Field-Programmable Gate Array*, en français plus ou moins « mer de blocs logiques reprogrammables »), plutôt destinés aux tests et aux petites quantités de production, ces puces étant à la fois très complètes (souvent 50% de leurs composants internes ne sont pas utilisés) et reprogrammables à l'infini (la plupart utilisent la technologie SRAM).
- Les ASIC (*Application-Specific Integrated Circuit*, en français « circuit intégré à application spécifique »), plutôt destinés à la production de masse. Ces puces sont construites sur mesure avec juste le bon nombre de blocs logiques et de fonctionnalités, pour minimiser les coûts de fabrication. Le coût de développement important est amorti par la quantité de puces produites et vendues.

Cette polyvalence d'usage (description, simulation, synthèse) s'exprime dans le code VHDL lui-même. Comme on va le découvrir tout au long de cette présentation, il y a de multiples façons d'écrire du VHDL.

2 LA GUERRE DES HDL

Comme son nom l'indique, le VHDL est un « HDL », un langage de description du matériel. Contrairement aux langages classiques (C, Java, Perl, ...), un HDL ne sert pas à contrôler l'ordinateur sur lequel il tourne. Il n'y a donc pas beaucoup de références aux fonctions fournies par le système d'exploitation (disques, fichiers, réseau, interface utilisateur, ...). C'est un univers à part, en vase clos, avec des contraintes très spécifiques au monde de la conception électronique, ce qui peut dérouter les informaticiens qui découvrent le VHDL.

Mais si le VHDL est le plus répandu en Europe, c'est le Verilog qui est majoritaire aux Etats-Unis. D'où la question à 128 bits : lequel de ces deux langages choisir et apprendre, lequel est le meilleur ?

Si vous connaissez les conflits historiques tels Windows contre Linux, Vi contre EMACS, Mac contre PC, KDE contre Gnome, en voici un autre d'une virulence comparable. L'esprit de cet antagonisme peut être décrit au moyen du passage suivant, que l'on retrouve sous différentes formes sur Usenet :

« Verilog a été conçu par des experts du hardware, sans réelle connaissance de comment écrire un langage de programmation, il a donc dû être bidouillé un bon moment avant d'être vraiment utile.

VHDL a été conçu par des experts des langages de programmation qui n'y connaissaient rien au monde du hardware, il a donc dû être bidouillé un bon moment avant d'être vraiment utile. »

Evidemment, derrière l'humour grinçant, la réalité est beaucoup plus complexe et souvent intranchable. Cela dépend bien sûr du type de projet et des goûts et besoins de chacun, car aucun langage n'est parfait dans tout...

L'approche choisie par la plupart des créateurs de logiciels de simulation et de synthèse a été de ne pas choisir, donc de supporter simultanément et également le VHDL et Verilog dans leurs outils de CAO. Ce qui n'aide pas vraiment à faire un choix.

L'autre critère est la zone géographique ou l'influence de l'environnement : certaines sociétés favorisent un HDL par rapport à l'autre. Mais objectivement, que peut-on dire ?

La syntaxe de Verilog est moins lourde, un peu plus proche du C, mais sans jamais en être (ce qui déroute aussi les débutants qui croient avoir affaire à un langage de programmation classique). Cela rend l'écriture d'un code plus rapide, ce qui plait souvent aux managers car ils pensent que cela augmente la productivité.

Mais il est indéniable que le VHDL est plus puissant. Il y a bien plus de choses à connaître et comprendre, mais ce sont d'autant plus de moyens de décrire un circuit en fonction de l'approche du moment. Ce n'est pas comparable aux langages classiques, où une opération est une opération...

Les contraintes qu'impose le VHDL sont en fait des structures qui deviennent indispensables lorsque le code grandit et doit s'adapter à de nouvelles utilisations. Ce langage qui impose des règles claires de gestion du code est donc favorable aux très gros projets. C'est d'ailleurs pour cela qu'il est dérivé du langage ADA, qui est lui aussi issu d'une initiative du DARPA.

3 PREMIÈRE APPROCHE DU LANGAGE

3.1 Description d'un circuit électronique

Pour présenter les caractéristiques d'un langage, rien de tel qu'un exemple concret. J'ai choisi de reproduire un circuit intégré standard, le 74HC595, en partant de sa description fournie par les fabricants. Nous pourrons ainsi aborder de nombreuses notions et techniques, employées très couramment dans tous les autres systèmes électroniques : les instanciations, les circuits séquentiels, la logique à 3 états, ...

La finalité de ce circuit est le décalage des bits, permettant de transformer un signal série en 8 signaux parallèles. Cette opération est réalisable avec un registre à décalage. Autour de cet usage, des fonctionnalités ont été ajoutées afin d'étendre le circuit à plus d'applications.

Le composant de base de ce circuit est la bascule D, une porte logique qui recopie son entrée (D) dans sa sortie (Q) à chaque coup d'horloge (C). Dans certains cas, une entrée « reset » est ajoutée, permettant de remettre la bascule à zéro. Attention, il ne faut pas confondre bascule D et latch. Le latch recopie le bit de l'entrée vers la sortie, même si celui-ci change, tant que l'horloge est à 1. Tandis que la bascule recopie l'entrée dans la sortie, en se basant uniquement sur l'état de l'entrée au moment du front montant de l'horloge. Quand l'horloge monte à 1, elle recopie l'entrée dans la sortie puis reste figée sur cette donnée jusqu'au prochain front montant d'horloge.

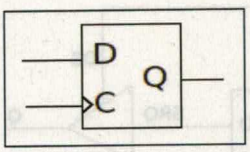


Figure 1 : Le symbole d'une bascule D

Il y a trois couches de composants dans le 74HC595 :

- Au premier niveau, il y a le registre à décalage lui-même : 8 bascules D chaînées les unes derrière les autres reçoivent le signal SER (SER est l'abréviation de *serial*, série en anglais). Ce signal est la donnée série elle-même. La sortie de chaque bascule est redirigée d'une part vers l'entrée de la bascule suivante et d'autre part vers un second niveau de bascules D. Ce premier niveau est animé par l'horloge SRCLK (SR comme *Shift Register* et CLK comme *clock*). Chaque bascule possède aussi une entrée « reset », utilisant le signal SRCLR (CLR est l'abréviation de *clear*, nettoyer en anglais). Enfin, au bout de cette chaîne, une sortie (QH) permet de récupérer le signal série initial afin de chaîner d'autres registres à décalage et augmenter la longueur à volonté.

- Le second niveau est une simple mémorisation de l'état des sorties des 8 bascules précédentes, grâce à 8 nouvelles bascules D. Ce niveau est géré par une horloge indépendante : RCLK (R comme registre). Ce niveau sert de tampon pour mémoriser les données parallèles jusqu'à leur exploitation. Ainsi, la sortie ne bouge pas lorsque l'on décale les données dans le circuit.

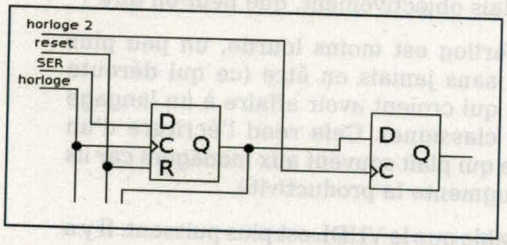


Figure 3 : Un assemblage de deux bascules D

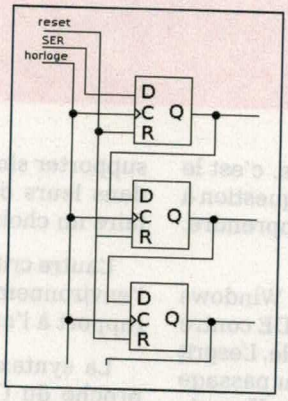


Figure 2 : Une chaîne de bascules D réalisant un registre à décalage

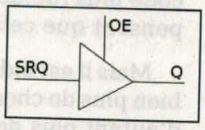


Figure 4 : Une sortie à 3 états

- Le troisième niveau prend en entrée d'une part les sorties de chacune des secondes bascules D et d'autre part le signal OE (*Output Enable*, sortie active en anglais). Ce dernier niveau, composé de 8 « sorties à trois états », permet d'isoler électriquement les sorties du circuit afin de pouvoir éventuellement partager un bus de données, par exemple. Lorsque le signal OE est à zéro, toutes les sorties de ce niveau sont en haute impédance, un état flottant qui n'est ni 0, ni 1 et ne peut être lu directement par un composant électronique. Lorsque OE est à 1, la sortie est égale au signal de sortie des bascules D, c'est-à-dire que le niveau se transforme en un simple fil.

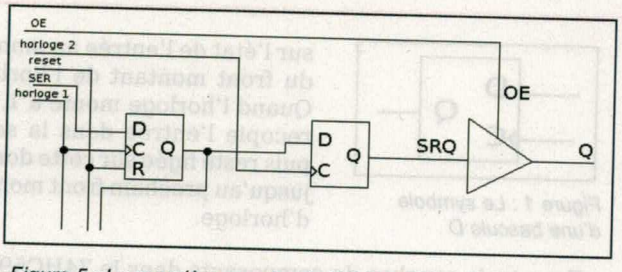


Figure 5 : La première couche du 74HC595

3.2 Première traduction en VHDL

Cette longue description peut être traduite en VHDL avec des unités de conception. Une unité de conception est un couple entité/architecture décrivant une « brique » du circuit. Chacun des composants de ce circuit sera décrit par une unité de conception. De façon à pouvoir réutiliser facilement le code, nous allons créer un fichier pour chaque unité de conception.

3.2.1 Préambule, quelques repères de grammaire

Il aurait été impensable de commencer à écrire du code sans rappeler quelques règles de base.

- Les commentaires :

En VHDL, les commentaires commencent par deux tirets (`--`) et s'écrivent sur une seule ligne. Voici un bon exemple :

```
-- commentaire valide
-- sur deux lignes
```

- Les points-virgules :

De façon générale, chaque instruction se finit par un point-virgule. On en utilise aussi dans les déclarations de port, pour séparer les différents types de signaux.

- Les identificateurs :

Ils doivent toujours commencer par une lettre et peuvent s'écrire invariablement en minuscules ou en majuscules. Par exemple, `Nom_de_variable` peut s'écrire `NOM_DE_VARIABLE` ou encore `nom_de_variable`. Ils ne doivent pas contenir d'espaces.

- Les flèches (c'est-à-dire `<=` ou `=>`) :

Elles servent à deux choses :

- `<=` est une assignation (un signal est copié dans un autre), par exemple `A<=B` signifie « B est recopié dans A ».
- `=>` est une association (pour port map, par exemple), `A=>B` signifie « A est associé à B », c'est-à-dire que A est instantanément égal à B.

3.2.2 Préambule, les types de données en VHDL

Il existe actuellement trois classes d'objets en VHDL : les constantes, les variables et les signaux. Les deux premiers sont communs aux langages de programmation classiques. Le signal est quant à lui un objet particulier, qui représente un fil statique dans un circuit ou entre deux circuits. Il existe du début à la fin de la simulation et ne peut être modifié dynamiquement. Ces objets peuvent être de quatre catégories de types différents :

- Les types scalaires (valeur individuelle) :
  - nombres flottants (*float*) ;
  - physiques (*time*) ;
  - nombre entiers (*integer*) ;
  - énumérés (*BIT, character, boolean, ...*).
- Les types composites (groupes de valeurs) :
  - tableau (*array* : vecteur de bits (un octet est un vecteur de 8 bits), chaîne de caractères (par exemple « hello world »)) ;
  - record.
- Le type pointeur (*ACCESS*) ;
- Le type fichier (*file*).

Bien sûr, chaque type possède un certain nombre de valeurs possibles. Par exemple, le type BIT ne peut prendre que deux valeurs : 0 ou 1. Il est également possible de créer ses propres types ou d'en importer de nouveaux. Par exemple, la bibliothèque IEEE permet d'inclure le type `std_logic`, un type que nous allons bientôt examiner.

3.2.3 Première étape : bibliothèques et paquetages

Un ensemble d'unités de conception peut être réuni dans une bibliothèque, contenant elle-même des paquetages, contenant eux mêmes des sous-programmes. Certaines bibliothèques sont standards et il est aussi possible de créer nos propres bibliothèques. Celles-ci permettent un travail organisé, seul ou en équipe, plus facile à développer quand il dépasse une certaine taille (ce qui arrive très vite) et adapté au terrain électronique (multiplicité des composants, parallélisme et/ou simultanéité entre les opérations, ...).

Le paquetage **std_logic** a été créé pour étendre les capacités du VHDL de base, en définissant le type **std_logic**. Celui-ci permet (entre autres) le support de l'état de haute impédance en plus des états booléens classiques que sont 0 et 1. Ce troisième état (défini par la lettre **'Z'** dans le paquetage) est important pour porter le code dans un environnement matériel et sortir de la « simple » simulation. Il permet, dans notre cas, de coder des sorties à trois états. Ce paquetage fait partie de la bibliothèque IEEE, définie par l'IEEE (*Institute of Electrical and Electronics Engineers*). En résumé, pour pouvoir s'offrir des sorties à trois états, il suffit de l'inclure en amont de chaque unité de conception qui en a besoin et de modifier tous les signaux de type BIT en signaux de type **std_logic**. Par défaut, un signal initialisé comme étant du type **std_logic** aura pour valeur **'U'** : *uninitialized*. Cela permet de détecter un défaut d'initialisation d'un circuit lors d'une simulation.

```
-- un extrait du paquetage std_logic :

TYPE std_ulogic IS (
  'U',  -- Uninitialized
  'X',  -- Forcing  Unknown
  '0',  -- Forcing  0
  '1',  -- Forcing  1
  'Z',  -- High Impedance
  'W',  -- Weak     Unknown
  'L',  -- Weak     0
  'H',  -- Weak     1
  '-'   -- Don't care
);

-- inclusion du paquetage std_logic :

library IEEE;
use ieee.std_logic_1164.all;
```

3.2.4 Deuxième étape : entités et architectures

L'entité est une « boîte noire » attribuant à l'unité de conception des liens avec l'extérieur (les entrées/sorties). Chaque entrée/sortie déclarée ici, représente un fil qui entre ou sort du circuit.

Chaque entité peut avoir une ou plusieurs architectures. Elle décrit le fonctionnement interne de l'unité de conception sous forme d'algorithmes, de séquences d'opérations logiques, figées ou non dans le temps. Avoir plusieurs architectures pour un seul modèle permet de changer de support matériel, de vision sur l'opération effectuée, de style d'écriture, ...

```
-- exemple complet de l'unité de
-- conception de la sortie à trois états :

library IEEE;
use ieee.std_logic_1164.all;

 -- déclaration de l'entité sortie_3_etats
entity sortie_3_etats is
 port (
   OE,T_D : in std_logic;
   T_Q : out std_logic
   );

-- fin de la description de l'entité
end sortie_3_etats;

-- déclaration de l'architecture beta_S3E
architecture beta_S3E of sortie_3_etats is
begin
  -- T_D est recopié dans T_Q quand OE est égal à zéro.
  T_Q <= T_D when (OE = '0')
  -- sinon Z est recopié dans T_Q.
  else 'Z';
  -- comme toutes les valeurs de type std_logic,
  -- Z s'écrit entre simples quotes.

-- fin de la description de l'architecture
end beta_S3E;
```

3.2.5 Troisième étape : process

Au sein d'une architecture, il est parfois utile de créer un processus. En effet, celui-ci possède des caractéristiques intéressantes. Par exemple, un processus s'exécute de son mot-clé **begin** à son mot-clé **end process**, avant de retourner à son mot-clé **begin**, constituant ainsi une boucle sans fin. De plus, un processus permet d'inclure des instructions séquentielles telles que **if**, **else** ou **for**, qui permettent plus de choses qu'une écriture par assignations directes (telles que le **if ... then ... else ... endif**).

```
-- architecture avec process :

library IEEE;
use ieee.std_logic_1164.all;

 --- bascule D sans reset
entity bascule_D is
  port (
    R_D,RCLK:in std_logic;
    R_Q:out std_logic
    );
end bascule_D;

architecture beta_D of bascule_D is
begin
 -- le code qui suit sera exécuté à
 -- chaque fois que le signal RCLK
 -- changera d'état.

 process(RCLK)
  begin
   -- si un évenement se produit sur
   -- RCLK (RCLK'event) et que celui-
   -- ci est égal à 1 (RCLK = '1')
   if RCLK'event and (RCLK = '1')
    -- alors l'entrée est recopiée dans la sortie
    then R_Q <= R_D;

  -- un if se termine toujours par
  -- cette ligne :
  end if;
 -- fin du process
 end process;
end beta_D;
```

Il va être nécessaire de différencier les bascules D sans signal de reset (**bascule_D**) des bascules D avec reset (**bascule_D_R**), celles-ci incluant un signal supplémentaire. Le code suivant, dérivé du précédent, est celui qui peut être créé pour décrire une bascule avec reset.

```
-- bascule D avec reset :

library IEEE;
use ieee.std_logic_1164.all;

entity bascule_D_R is
 port (
    SR_D,SRCLK,SRCLR:in std_logic;
    SR_Q:out std_logic
    );
end bascule_D_R;

architecture beta_R of bascule_D_R is
begin

process(SRCLR, SRCLK)
 begin
```

```
    -- la condition de reset est testée avant les autres,
    -- donc elle a priorité sur les autres
    if (SRCLR = '1') then
        SR_Q <= '0';
    elsif SRCLK'event and SRCLK = '1' then
        SR_Q <= SR_D;
    end if;

 end process;
end beta_R;
```

3.2.6 Quatrième étape : création de composants

Pour éviter de devoir créer 16 unités « bascules D » et 8 unités « sorties à trois états », nous allons créer un composant pour chacun de ces objets. En effet, un composant peut être réutilisé autant de fois qu'on le veut, on dit qu'on crée une « instance » du composant.

Ce code sera inclus dans l'unité de conception traitant de la puce entière, écrit un peu plus loin dans l'article.

```
-- déclaration d'un composant :
component bascule_D_R is
  port (
    RCLK, R_D : in std_logic;
    R_Q : out std_logic);
end component;
```

Les entrées et sorties du composant et de l'instance du composant (ici appelée BR1) seront reliées entre elles à l'aide du code suivant, qui fait en même temps la déclaration de l'instance du composant et le câblage de celle-ci :

```
-- instance de composant :
BR1 : bascule_D port map (RCLK=>RCLK, R_D=>SR_Q1, R_Q=>R_Q1);
```

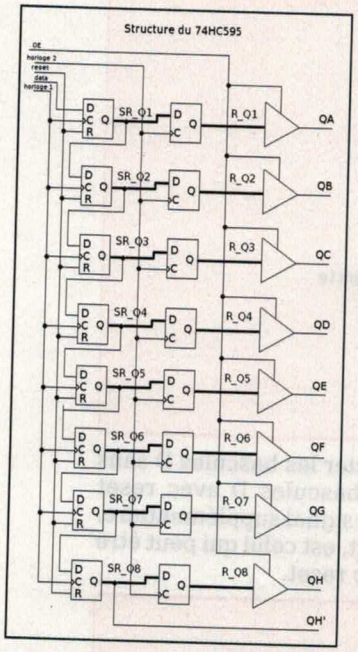


Figure 6 : La totalité du 74HC595

Il y aura donc respectivement :

- 8 instances du composant « bascule D avec reset » ;
- 8 instances du composant « bascule D sans reset » :
- 8 instances du composant « sorties à trois états ».

3.2.7 Cinquième (et dernière) étape : utilisation des composants et création de la puce complète

La dernière unité de conception qui sera créée est celle décrivant le circuit 74HC595 dans son ensemble.

L'entité contiendra la liste des entrées et sorties du circuit (les 8 sorties parallèles, la sortie série et les deux horloges, le signal de reset, l'entrée série et le signal OE). Notre architecture contiendra quant à elle les 3 composants (les 2 bascules D et la sortie à trois états), l'instanciation et le câblage de ceux-ci, réalisés grâce au mot-clé **port map**.

```
-- le code de l'unité de conception 74HC595 :
library IEEE;
use ieee.std_logic_1164.all;

-------------- 74HC595 --------------
entity SN74HC595 is
  port (
    SER,SRCLK,SRCLR,RCLK,OE : in std_logic;
    QA,QB,QC,QD,QE,QF,QG,QH,QHP : out std_logic
    );
end SN74HC595;

architecture desc_str of SN74HC595 is

-- liste des composants :

component bascule_D_R is
  port (SR_D, SRCLK, SRCLR : in std_logic; SR_Q : out std_logic);
end component;

component bascule_D is
  port (RCLK, R_D : in std_logic; R_Q : out std_logic);
end component;

component sortie_3_etats is
  port (OE, T_D : in std_logic; T_Q : out std_logic);
end component;

signal SR_Q1, SR_Q2, SR_Q3, SR_Q4, SR_Q5, SR_Q6, SR_Q7, SR_Q8,
R_Q1, R_Q2, R_Q3, R_Q4, R_Q5, R_Q6, R_Q7, R_Q8 : std_logic;

begin

-- cablage des composants

BSR1 : bascule_D_R port map (SR_D=>SER,  SRCLK=>SRCLK, SRCLR=>SRCLR, SR_Q=>SR_Q1);
BSR2 : bascule_D_R port map (SR_D=>SR_Q1, SRCLK=>SRCLK, SRCLR=>SRCLR, SR_Q=>SR_Q2);
BSR3 : bascule_D_R port map (SR_D=>SR_Q2, SRCLK=>SRCLK, SRCLR=>SRCLR, SR_Q=>SR_Q3);
BSR4 : bascule_D_R port map (SR_D=>SR_Q3, SRCLK=>SRCLK, SRCLR=>SRCLR, SR_Q=>SR_Q4);
BSR5 : bascule_D_R port map (SR_D=>SR_Q4, SRCLK=>SRCLK, SRCLR=>SRCLR, SR_Q=>SR_Q5);
BSR6 : bascule_D_R port map (SR_D=>SR_Q5, SRCLK=>SRCLK, SRCLR=>SRCLR, SR_Q=>SR_Q6);
BSR7 : bascule_D_R port map (SR_D=>SR_Q6, SRCLK=>SRCLK, SRCLR=>SRCLR, SR_Q=>SR_Q7);
BSR8 : bascule_D_R port map (SR_D=>SR_Q7, SRCLK=>SRCLK, SRCLR=>SRCLR, SR_Q=>SR_Q8);

QHP <= SR_Q8;

BR1 : bascule_D port map (RCLK=>RCLK, R_D=>SR_Q1, R_Q=>R_Q1);
BR2 : bascule_D port map (RCLK=>RCLK, R_D=>SR_Q2, R_Q=>R_Q2);
BR3 : bascule_D port map (RCLK=>RCLK, R_D=>SR_Q3, R_Q=>R_Q3);
BR4 : bascule_D port map (RCLK=>RCLK, R_D=>SR_Q4, R_Q=>R_Q4);
BR5 : bascule_D port map (RCLK=>RCLK, R_D=>SR_Q5, R_Q=>R_Q5);
BR6 : bascule_D port map (RCLK=>RCLK, R_D=>SR_Q6, R_Q=>R_Q6);
BR7 : bascule_D port map (RCLK=>RCLK, R_D=>SR_Q7, R_Q=>R_Q7);
BR8 : bascule_D port map (RCLK=>RCLK, R_D=>SR_Q8, R_Q=>R_Q8);

STE1 : sortie_3_etats port map (OE=>OE, T_D=>R_Q1, T_Q=>QA);
STE2 : sortie_3_etats port map (OE=>OE, T_D=>R_Q2, T_Q=>QB);
STE3 : sortie_3_etats port map (OE=>OE, T_D=>R_Q3, T_Q=>QC);
STE4 : sortie_3_etats port map (OE=>OE, T_D=>R_Q4, T_Q=>QD);
STE5 : sortie_3_etats port map (OE=>OE, T_D=>R_Q5, T_Q=>QE);
STE6 : sortie_3_etats port map (OE=>OE, T_D=>R_Q6, T_Q=>QF);
STE7 : sortie_3_etats port map (OE=>OE, T_D=>R_Q7, T_Q=>QG);
STE8 : sortie_3_etats port map (OE=>OE, T_D=>R_Q8, T_Q=>QH);

end desc_str;
```

CONCLUSION

Ainsi s'achève cet exemple d'utilisation du VHDL, qui sera simulé dans mon prochain article. Nous y étudierons (entre autres) GHDL, un compilateur/simulateur unixien sous GPL et verrons comment intégrer cette description dans un petit FPGA. Signalons aussi le prochain article de cette série : « le VHDL pour ceux qui ont débuté », permettant d'approfondir les notions abordées ici, en poursuivant le même exemple. Je tiens à remercier tout particulièrement whygee, mon gourou de l'électronique.

Liens

- Pour ceux qui veulent aller plus loin : http://comelec.enst.fr/hdl/vhdl_intro.html
- Pour télécharger le *datasheet* du 74HC595 : http://www.ti.com/lit/gpn/sn74hc595
- L'archive du code source VHDL final : http://ygllo.com/~llo/vhdl/

Auteur : Laura Bécognée