La mise en oeuvre de Linux pour l'embarqué

Patrice Kadionik, Maître de Conférences à l'ENSEIRB (kadionik@enseirb.fr)

Mars 2006

Introduction

Bienvenue dans le monde de l'embarqué et plus précisément de Linux embarqué (le retour) !

Dans ce Hors Série résolument orienté pratique, cet article d'introduction se propose de faire la revue de ce que vous, lecteur, avez besoin pour pouvoir vous lancer dans cette aventure.

L'environnement de développement croisé sera présenté dans un premier temps. Les différentes étapes de développement d'une application Linux embarqué seront aussi décrites. Enfin, un exemple pratique de mise en oeuvre sera donné pour illustrer le tout.

Linux embarqué : l'environnement de développement croisé à mettre en oeuvre

Pour pouvoir « faire » du Linux embarqué, il faut d'abord mettre en place un environnement de développement croisé.

On a besoin pour cela de deux éléments indispensables :

- le PC de développement sous Linux. C'est la machine hôte ou *host*. Généralement, un compilateur C/C++ est préalablement installé. C'est un compilateur croisé capable de générer du code exécutable par le système embarqué à développer. On utilise en pratique le compilateur C/C++ GNU *gcc*.
- Le système embarqué. C'est la machine cible ou carte cible ou *target*. Il exécutera le noyau Linux, les utilitaires et les applications Linux embarqué développées.

La machine hôte et la carte cible sont reliées au minimum par l'intermédiaire d'une liaison série pour le téléchargement, la programmation et le « débogage » de la cible. La liaison série sert aussi sous Linux comme port par défaut où sont envoyées les traces du noyau Linux embarqué.

Malheureusement, cette liaison, même programmée à 115,2 kb/s, est trop lente pour servir comme moyen de téléchargement rapide de fichiers de quelques Mo, ce que l'on a pour Linux embarqué. Cela peut prendre de quelques minutes à quelques dizaines de minutes.

On préfère donc utiliser une liaison Ethernet bien plus rapide : le téléchargement durera de quelques secondes à quelques dizaines de secondes. Cela implique d'avoir une connexion Ethernet présente sur la carte cible. Le moniteur/*bootloader* de la carte cible doit pouvoir supporter le téléchargement de fichiers par Ethernet. Pour cela, il intègre un client TFTP (*Trivial FTP*) fonctionnant au dessus du protocole de transport UDP qui est facilement « romable » (adaptable sur mémoire morte ou FLASH) du fait d'être un protocole de transport non connecté et sans état.

L'environnement de développement croisé typique est donc celui de la figure 1 :

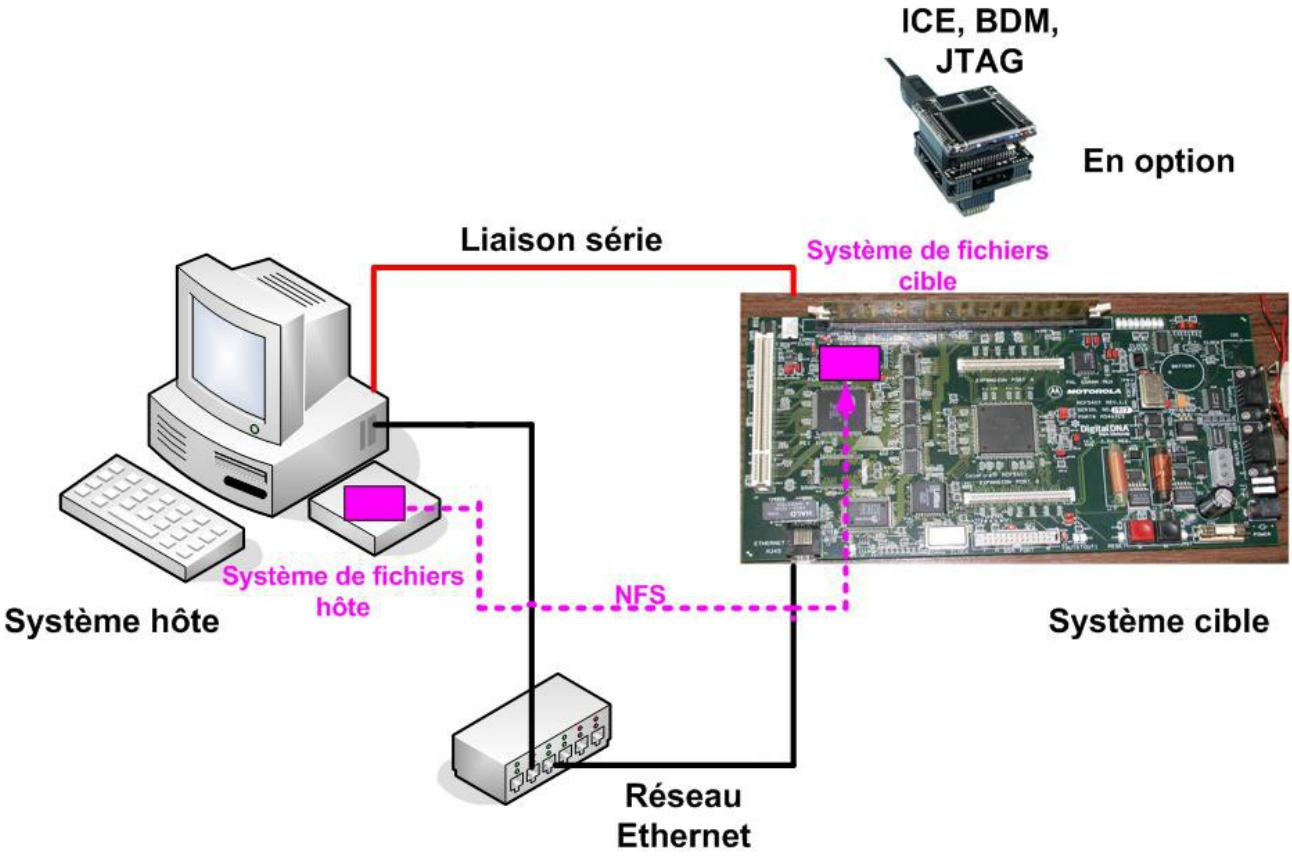


Figure 1 : Environnement de développement croisé pour Linux embarqué

La cible doit bien sûr pouvoir embarquer Linux. Pour cela, il faut que :

- Le processeur soit au moins un processeur 32 bits avec MMU (noyau Linux) ou sans MMU (noyau µClinux).
- Que la cible ait quelques Mo de mémoire RAM et ROM (FLASH pour sa facilité de reprogrammation).

Pour le débogage, on pourra en plus utiliser les dispositifs matériels supplémentaires suivants :

- Emulateur ICE (*In Circuit Emulator*). Cet équipement électronique est l'outil roi du débogage matériel et logiciel. Il « émule » complètement le processeur et en contrôle tous les signaux, ce qui permet d'avoir un débogage en Temps Réel du système, condition importante pour déverminer des routines d'interruption par exemple. Il permet aussi de déboguer des applications directement au niveau symbolique du fichier source C/C++. Ses principaux inconvénients sont le prix relativement important (quelques milliers d'Euros, voire plus) et le fait qu'il soit lié à un type de processeur donné.
- Debugger BDM (*Background Debug Module*). C'est un *debugger* OCD (*On Chip Debugger*) propriétaire de Freescale (anciennement Motorola). Ce *debugger* est intégré en dur dans le processeur et permet de l'espionner (accès mémoire) mais aussi de le contrôler (par accès aux registres du processeur par exemple). Il permet aussi de déboguer des applications directement au niveau symbolique du fichier source C/C++. Le *debugger* OCD est très bon marché car inclus par défaut dans le processeur mais il reste généralement moins performant qu'un émulateur ICE bien qu'il soit suffisant dans la majorité des cas.

- Debugger JTAG (*Joint Test Action Group*). Le JTAG est originellement une norme IEEE (1149.1) établie pour le test BIST (*Built In Self Testing*) des circuits électroniques des cartes de PC ! Cette interface matérielle a été reprise et complétée pour pouvoir être intégrée en dur dans un processeur pour former un *debugger* OCD. Comme précédemment, le *debugger* JTAG permet d'espionner et de contrôler le processeur. Une fonction usuelle du JTAG est la reprogrammation de la mémoire FLASH (avec un nouveau noyau Linux par exemple) ou la reprogrammation de circuits FPGA.

On pourra bien sûr aussi utiliser le célèbre GNU *debugger gdb* pour un débogage logiciel moins « fin » qu'avec les outils matériels précédents.

Les outils de débogage ICE, BDM et JTAG sont généralement fournis avec des outils propriétaires mais il existe des solutions pour les interfacer avec des outils libres de débogage orientés Linux. Il est possible par exemple de coupler le BDM avec *gdb* !

Enfin, un dernier point important apparaît sur la figure 1. L'application Linux embarqué doit être incluse dans le système de fichiers *root* utilisé par le noyau Linux embarqué. A chaque recompilation de l'application, cette opération doit être faite. Dans le cas de µClinux, durant la phase de développement, on se retrouve ainsi à exécuter à chaque recompilation de l'application le cycle infernal : reset de la cible, téléchargement du fichier contenant le système de fichiers *root* incorporant l'application et le noyau µClinux, *boot* du noyau Linux et tests de l'application.

La solution pour ne pas devoir exécuter à chaque fois ce cycle infernal synonyme de perte de temps, est d'utiliser la connectivité IP naturelle de Linux.

Pour cela, le PC hôte exporte une partie de son système de fichiers qui contiendra l'application Linux embarqué qui sera ensuite montée par NFS (*Network File System*) par la carte cible et apparaîtra donc localement dans le système de fichiers de celle-ci.

On peut aussi monter entièrement par NFS le système de fichiers *root* car c'est une option de *boot* du noyau !

On pourra mettre à profit l'usage d'un système de fichiers JFFS2 (*Journalling Flash File System* 2) en mémoire FLASH et accessible en lecture/écriture pour éviter ce cycle infernal. Il suffira alors de transférer l'application par le réseau (par *ftp* par exemple) dans le système de fichiers JFFS2 de la cible.

Linux embarqué : le développement d'une application

Développer une application Linux embarqué demande des prérequis du point de vue matériel puis logiciel.

Si l'on regarde les différentes étapes de développement d'un système électronique sous Linux embarqué, on peut distinguer les points suivants :

1. Développement du système électronique.
2. Développement du *bootloader* Linux.
3. Portage du noyau Linux.
4. Développement des drivers spécifiques.
5. Développement de l'application Linux embarqué.
6. Intégration du tout.

Etape 1 : Développement du système électronique

Il s'agit de la conception du système électronique mettant en oeuvre un processeur 32 bits avec ou sans MMU : processeurs ARM, ColdFire, x86, processeurs softcore Leon, OpenRisc, NIOS, MicroBlaze... Le matériel doit être opérationnel : accès mémoire, accès aux périphériques externes d'E/S, boot du processeur...

On pourra bien sûr acheter des cartes toutes faites pour s'affranchir de cette étape, ce qui est fortement conseillé ici. Il existe des cartes bon marché pour pouvoir expérimenter Linux embarqué :

- La carte ETRAX d'Acme Systems disponible chez Lextronic (http://www.lextronic.fr/fox/PP.htm).
- La carte ARM9 de Kwikbyte (http://www.kwikbyte.com/).
- Les cartes ARM7 ou ARM9 d'Eukrea (http://www.eukrea.com/).

On peut aussi utiliser des équipements électroniques plus standards sur lesquels Linux a été porté (à vos risques et périls) :

- Un PDA iPAQ.
- Une tablette NOKIA N770.
- Une console de jeux !

Etape 2 : Développement du *bootloader* Linux

Le bootloader est généralement un programme en mémoire non volatile (mémoire FLASH). Il prend en charge l'initialisation de base du système électronique, les autotests, et plus spécifiquement la décompression du noyau Linux et du système de fichiers *root* stockés généralement en mémoire FLASH ou téléchargés (liaison série ou réseau). Pour cela, on peut utiliser le *bootloader* Linux fourni avec le système si on l'a acheté. Si ce n'est pas un *bootloader* Linux mais un moniteur, il est toujours possible de lancer le noyau Linux.
Sinon, on pourra utiliser un *bootloader* Linux libre comme U-boot, RedBoot, Colilo…
Il faut bien sûr avoir le compilateur croisé GNU *gcc* correspondant à son type de processeur. Cette remarque est valable pour toutes les étapes suivantes...

Etape 3 : Portage du noyau Linux

Il faut récupérer les sources du noyau Linux (portage) correspondant au type de processeur du système électronique. Le noyau Linux a été porté sur pratiquement tous les processeurs 32 bits existants et cette étape ardue n'est plus à faire. Néanmoins, il faut éventuellement modifier les sources du noyau pour « coller » au hardware du système (changement des adresses de base des registres des périphériques d'E/S...).

Etape 4 : Développement des drivers spécifiques

Il est souhaitable de choisir des composants qui sont supportés par Linux lors de la phase de conception du système. Lors de la phase de configuration du noyau Linux, on choisira les drivers appropriés. On pourra éventuellement les modifier pour coller au mapping mémoire de la cible (voir étape 3).
Si l'on doit écrire un nouveau driver pour un matériel spécifique, il est préférable de partir d'un driver existant similaire.
Il faudra aussi faire le choix entre développer un driver statique lié au noyau ou un driver dynamique (module Linux) chargé en mémoire par le noyau à la demande. Cependant, la quasi-totalité des

drivers développés en dehors de l'arborescence du noyau l'est sous forme de modules.

Etape 5 : Développement de l'application Linux embarqué
Il est important de voir au préalable s'il n'existe pas déjà une application libre existante correspondant à son besoin. Dans ce cas, on pourra la porter sur sa cible. Dans le meilleur des cas, cela se traduira par une simple recompilation croisée. Sinon, il faudra la développer soi-même *from scratch*...

Etape 6 : Intégration du tout
C'est le bouquet final ! Il convient de vérifier si le matériel et le logiciel embarqué fonctionnent correctement comme cela a été spécifié par des tests d'intégration.

Il est aussi important d'évaluer les critères suivants :
- La taille mémoire consommée (RAM, FLASH) par le système sous Linux embarqué.
- La prise en compte de la faible empreinte mémoire des applications.
- Le chargement du noyau en RAM ou exécution XIP (*eXecute In Place*).
- Le système de fichiers *root* en mémoire RAM, en mémoire FLASH ou monté par NFS.
- Le type du système de fichiers utilisé : ext3, JFFS2 pour mémoire FLASH et adapté à l'embarqué.
- Les bibliothèques : compilation statique ou dynamique.
- Bibliothèque *libc* à faible empreinte mémoire : µClibc, newlib, dietlib...

Linux embarqué : un exemple pratique de mise en oeuvre

L'exemple pratique de mise en oeuvre va servir à illustrer ce qui a été dit précédemment. L'environnement de travail est le suivant :

- Le PC hôte est sous Linux avec une distribution Fedora Core.
- La carte cible est une carte d'évaluation Freescale M5407C3 à base de processeur ColdFire MCF5407 sans MMU (figure 2).
- La carte cible ColdFire exécute le noyau µClinux adapté aux processeurs sans MMU.

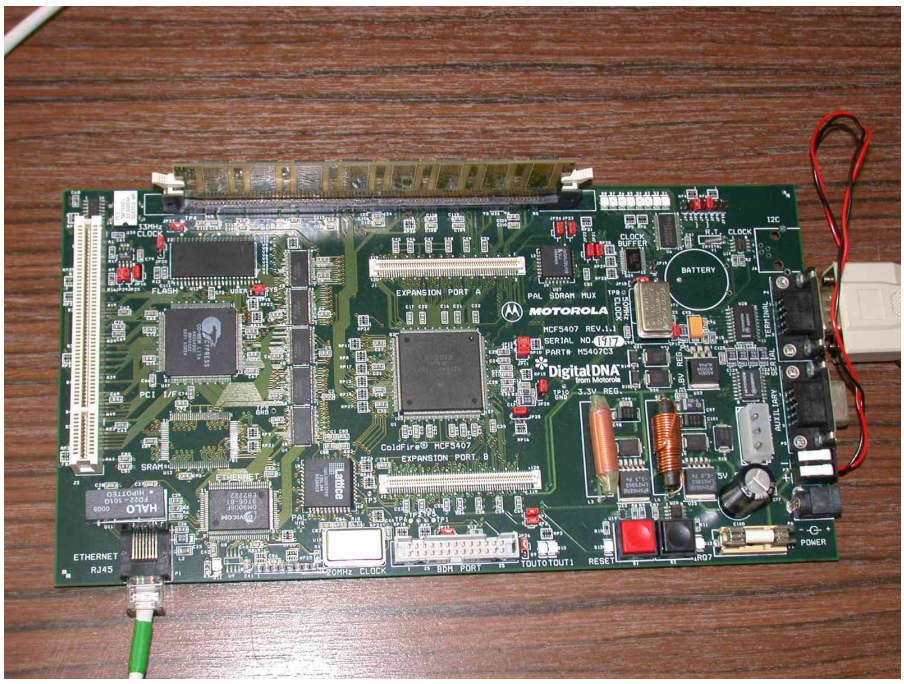


Figure 2 : Configuration du noyau µClinux/ColdFire

L'étape 1 ne s'applique pas ici. L'étape 2 ne s'applique pas non plus car on utilise le moniteur de la carte (*dBUG*). Les étapes 3 et 4 ne sont pas non plus applicables car la distribution µClinux contient le noyau µClinux adapté au processeur ColdFire et inclut le BSP (*Board Support Package*) de la carte cible. Seules l'étape 5 et l'étape 6 dans une moindre mesure, sont applicables ici.

Pour cela, il faut d'abord récupérer les archives sur le site µClinux/ColdFire :

- De la chaîne de compilation croisée pour le processeur ColdFire (fichier `m68k-elf-tools-xxxxxxxx.sh`).
- De la distribution de µClinux pour le processeur ColdFire (fichier `uClinux-dist-xxxxxxxx.tar.gz`).

La première chose est d'installer sur le PC hôte la chaîne de compilation croisée. En étant *root* :

```
# cd /
# ./m68k-elf-tools-20031003.sh
```

La deuxième chose est d'installer la distribution de µClinux/ColdFire dans son répertoire de travail en étant simple utilisateur :

```
$ cd
$ tar xvfz  uClinux-dist-20051110.tar.gz
$ cd uClinux-dist/
```

Cette distribution contient plusieurs éléments :

- Le noyau µClinux.
- Des bibliothèques *libc* adaptées comme µClibc.
- Un ensemble d'applications de base (applications *userland*) portées sous µClinux et servant à remplir le système de fichiers *root*.
- Des BSP correspondant à un certain nombre de cartes électroniques du commerce.
- Des scripts pour générer le noyau µClinux en fonction de sa carte cible et pour générer le système de fichiers *root* en fonction des applications *userland* choisies.

L'application Linux embarqué choisie comme illustration sera le célèbre *Hello World!* de K&R.

Pour cela, il faut modifier des fichiers de configuration pour prendre en compte la compilation croisée de son application *userland* `hello` (voir le HOWTO `Adding-User-Apps-HOWTO` inclus dans la distribution µClinux).

On crée le répertoire `hello` à partir du répertoire `ledcon` dans le *userland* sous `uClinux-dist/user` :

```
% cd uClinux-dist/user
% cp -r ledcon hello
```

On modifie le fichier `uClinux-dist/user/hello/Makefile` :

```
EXEC = hello
OBJS = hello.o

all: $(EXEC)

$(EXEC): $(OBJS)
	$(CC) $(LDFLAGS) -o $@ $(OBJS) $(LDLIBS)

romfs:
	$(ROMFSINST) /bin/$(EXEC)

clean:
	-rm -f $(EXEC) *.elf *.gdb *.o
```

On crée le fichier source C `uClinux-dist/user/hello/hello.c` :

```
#include <stdio.h>


main() {
printf("Hello World from uClinux!\n");
exit(0);
}
```

On modifie le fichier `uClinux-dist/user/Makefile` par ajout de la directive :

```
dir_$(CONFIG_USER_HELLO_HELLO)      += hello
```

On modifie le fichier `uClinux-dist/config/config.in` par ajout de la directive dans le menu `Miscellaneous Applications` :

```
bool    'hello'  CONFIG_USER_HELLO_HELLO
```

On génère ensuite le noyau µClinux ainsi que le système de fichiers *root* :

```
% make xconfig
```

Une série de fenêtres apparaît pour :

1. Choisir sa carte cible donc le BSP.
2. Configurer le noyau µClinux pour processeur ColdFire (noyau 2.0, 2.4 ou 2.6).
3. Choisir ses applications *userland*. On validera bien sûr la compilation de son application `hello`.

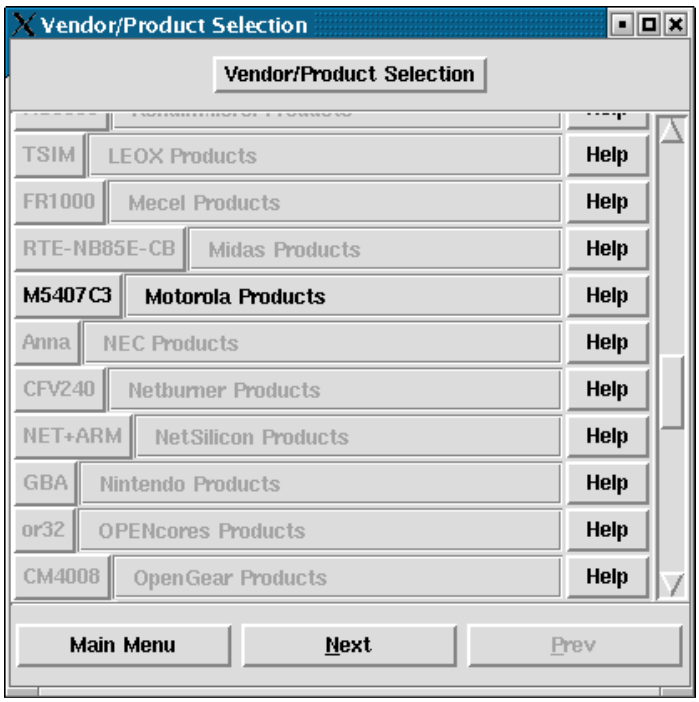


Figure 3 : Choix du BSP

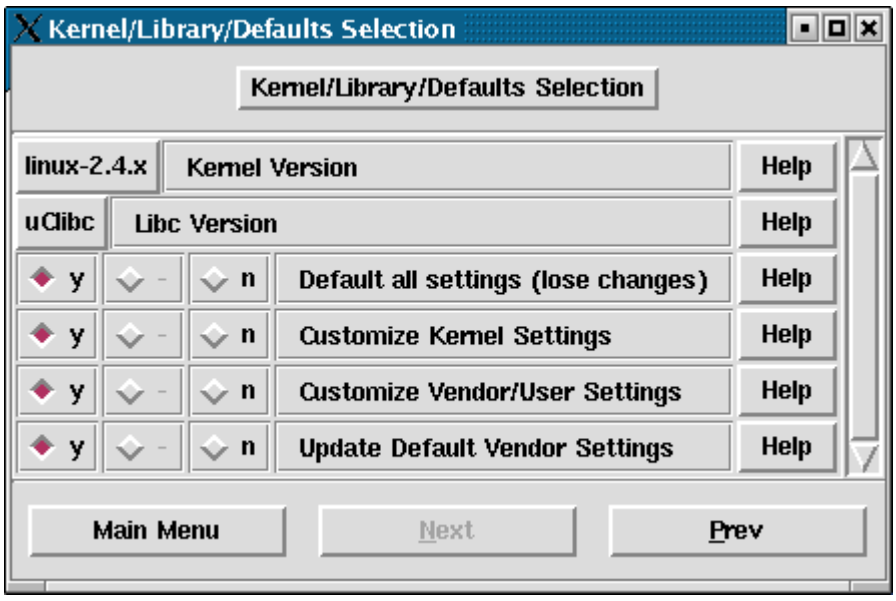


Figure 4 : Choix de la version du noyau µClinux/ColdFire (2.4 ici) et de la bibliothèque *libc (*µClibc ici)

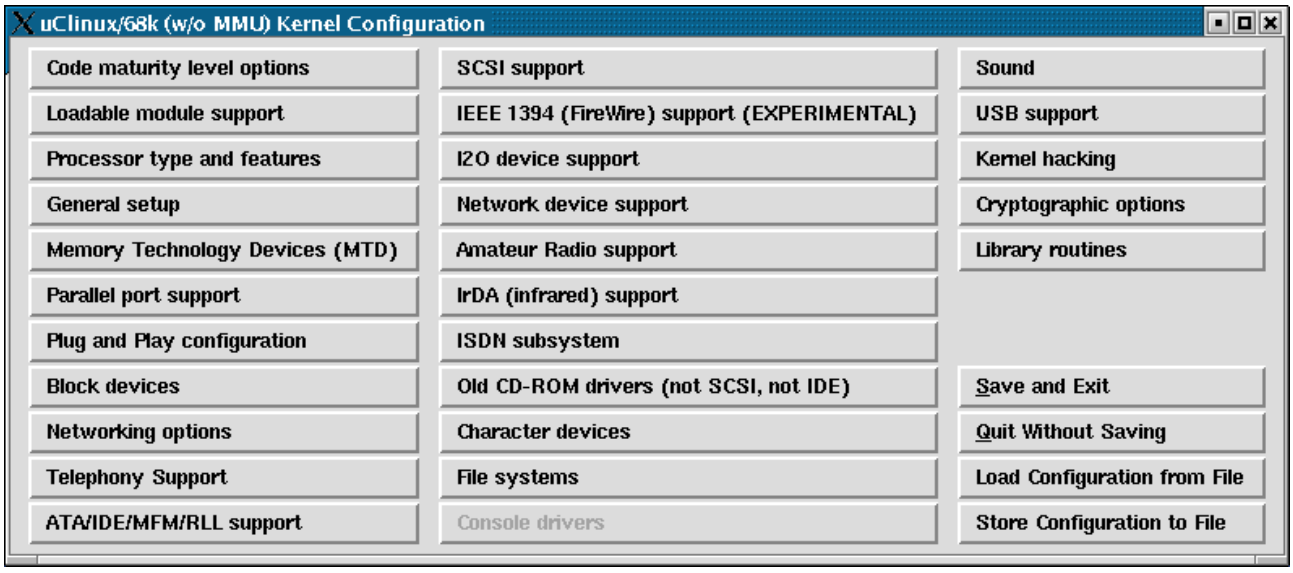


Figure 5 : Configuration du noyau µClinux/ColdFire

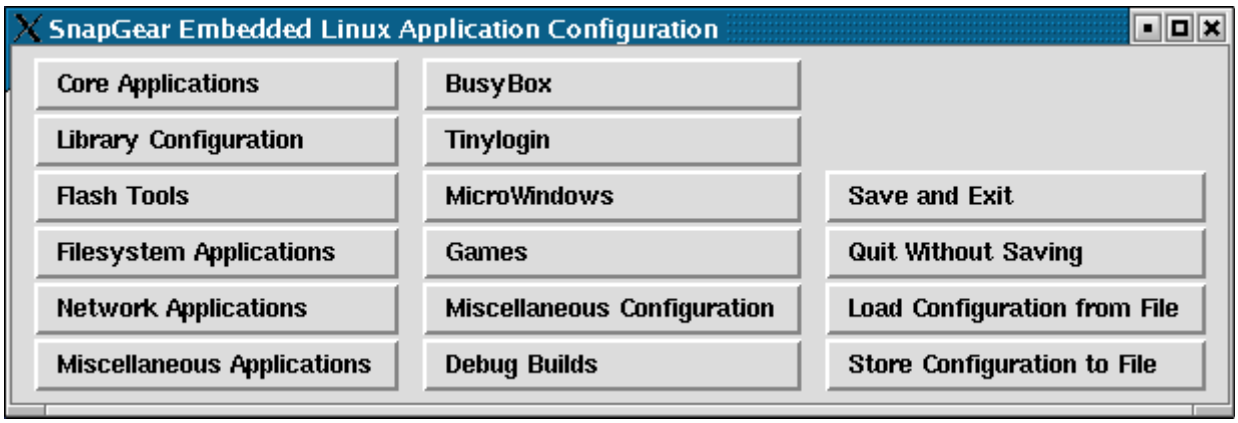


Figure 6 : Choix des applications *userland*

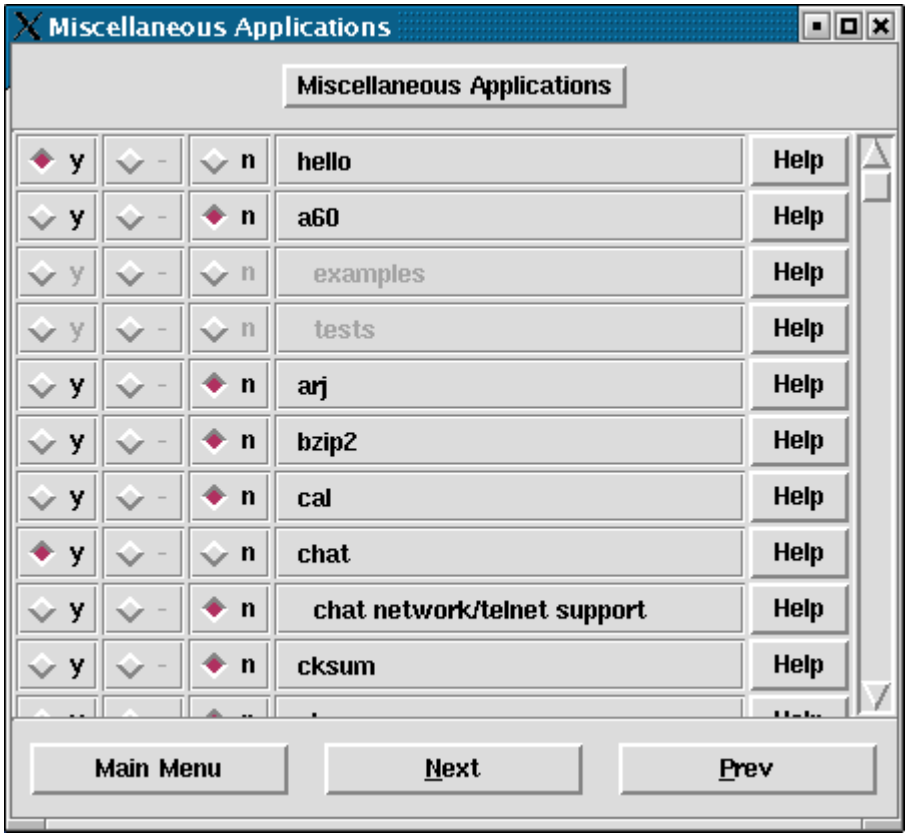


Figure 7 : Validation de la compilation croisée de l'application `hello`

On a ensuite la phase de compilation du noyau et des applications *userland* :

```
$ make dep
$ make
```

Un fichier binaire `image.bin` contenant le noyau µClinux ainsi que le système de fichiers *root* est placé sous `/tftpboot` pour pouvoir être téléchargé en RAM de la carte cible ColdFire via le réseau Ethernet avec le protocole TFTP.

Pour cela, on se connecte à la carte par le port série en utilisant l'outil *minicom* :

```
$ minicom
```

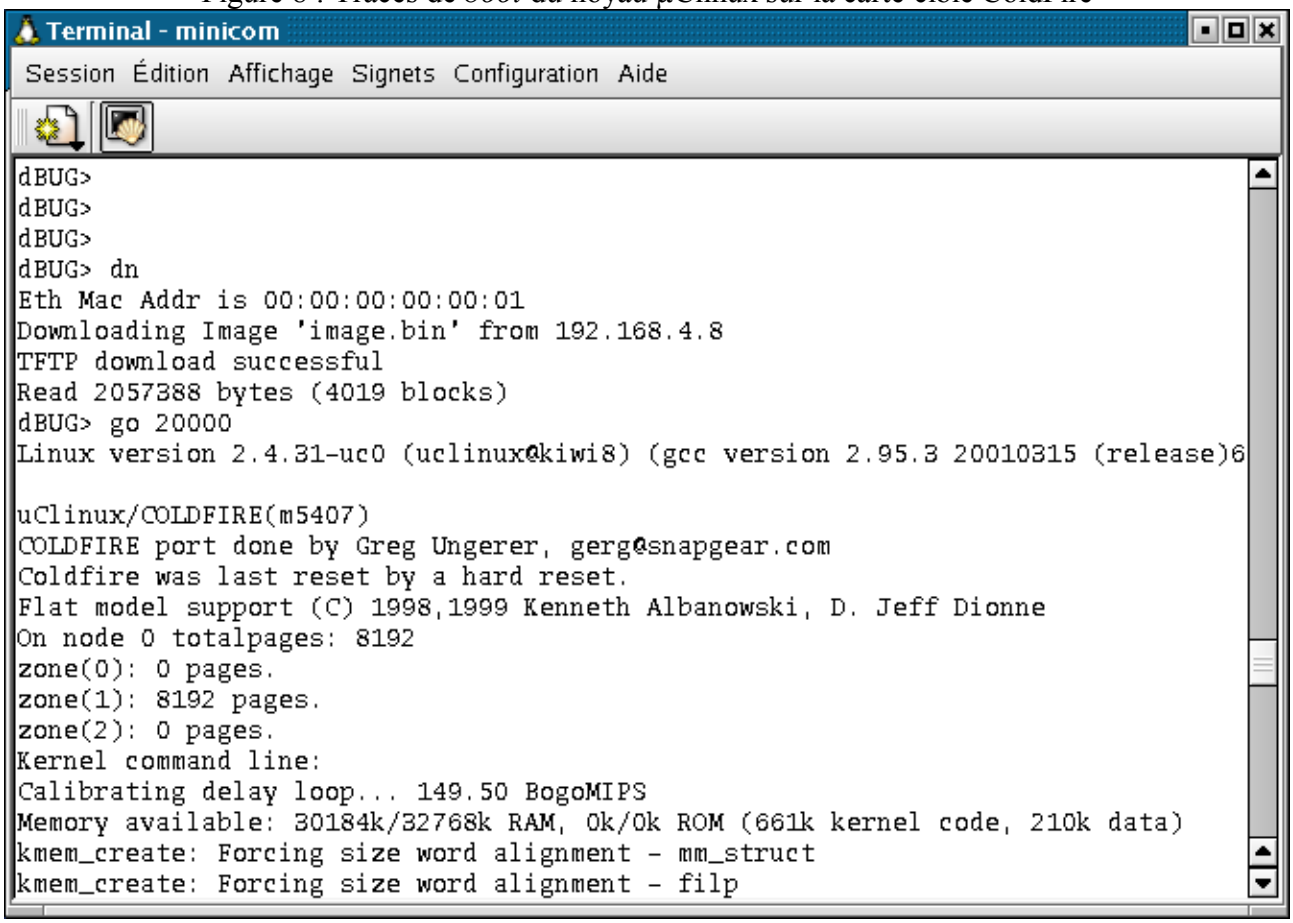
Depuis *minicom*, on télécharge le fichier `image.bin` en utilisant le moniteur *dBUG* de la carte cible :

```
dBUG> dn image.bin
```

Puis on lance le noyau µClinux/ColdFire :

```
dBUG> go 20000
```

Figure 8 : Traces de *boot* du noyau µClinux sur la carte cible ColdFire



On remarquera sur les traces précédentes que la taille du noyau µClinux avec ses données ne fait que 880 Ko, ce qui est faible et parfaitement romable !

Il ne reste plus qu'à tester son application `hello` sur la carte cible :

```
/> hello
```

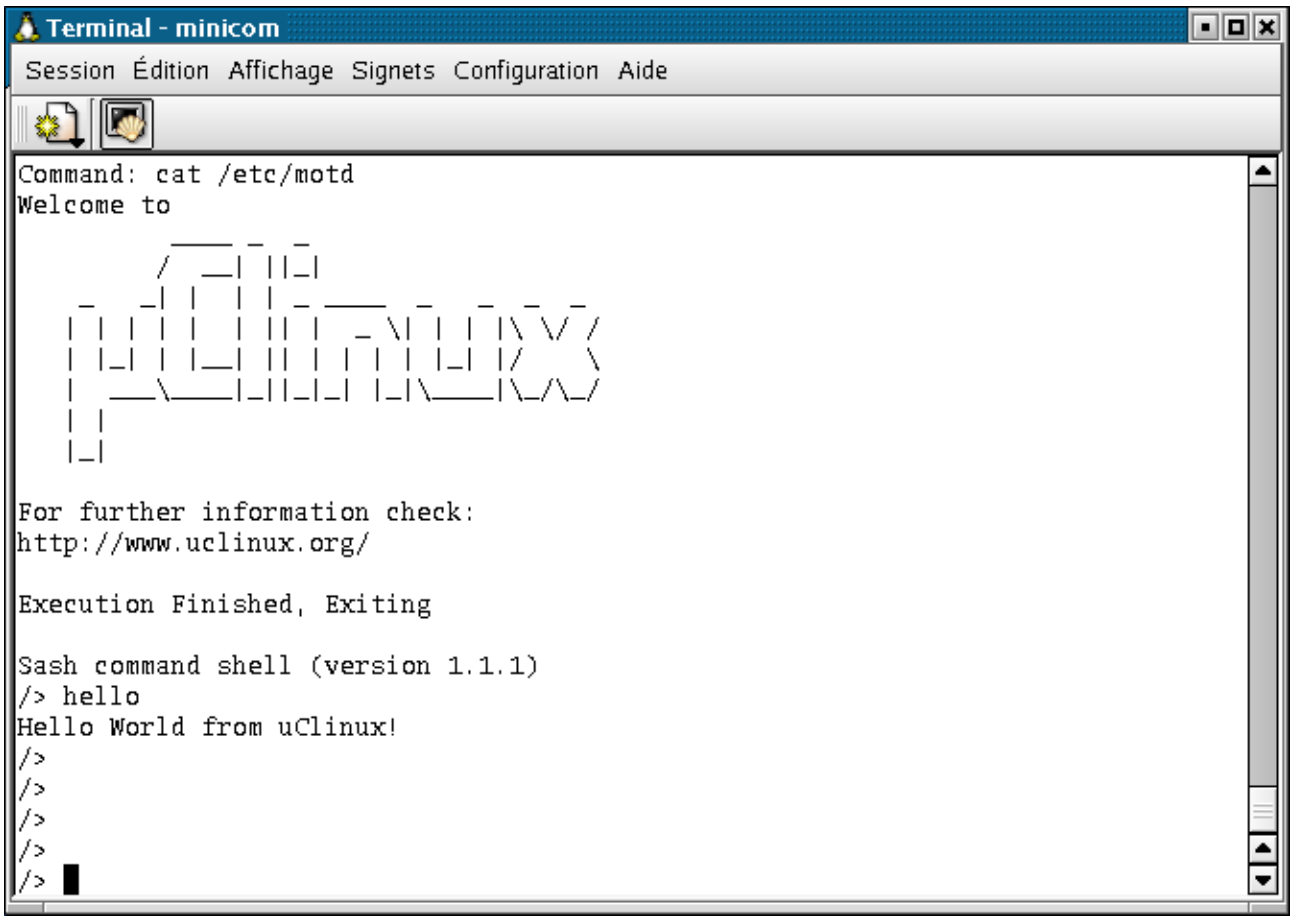


Figure 9 : Test du programme `hello` sur la carte cible ColdFire

Pour le débogage d'une application µClinux, on peut utiliser les possibilités de débogage logiciel par le réseau de *gdb*.

Pour cela, on reprend l'application `hello` modifiée pour cela :

```
#include <stdio.h>

unsigned int i;
unsigned int j;

main() {

j = 0;

for(i=0; i<10; i++)
     j++;

printf("Hello World from uClinux!\n");

exit(0);
}
```

Côté carte cible, après configuration de l'interface réseau (192.168.4.101 ici), on lance l'application

`gdbserver` sur un numéro de port d'écoute TCP (3000 ici) :

```
/> gdbserver :3000 /bin/hello
```

Côté PC hôte, on utilise l'outil de débogage graphique DDD (*Data Display Debugger*) couplé à *gdb* pour le processeur ColdFire :

```
$ cd
$ cd uClinux-dist/user/hello
$ ddd -debugger m68k-bdm-elf-gdb hello.gdb
```

Sous DDD, au niveau de la ligne de commande *gdb*, on exécute la commande :

```
(gdb) target remote 192.168.4.101:3000
```

Un exemple de session de débogage est illustré par les figures 10 et 11.

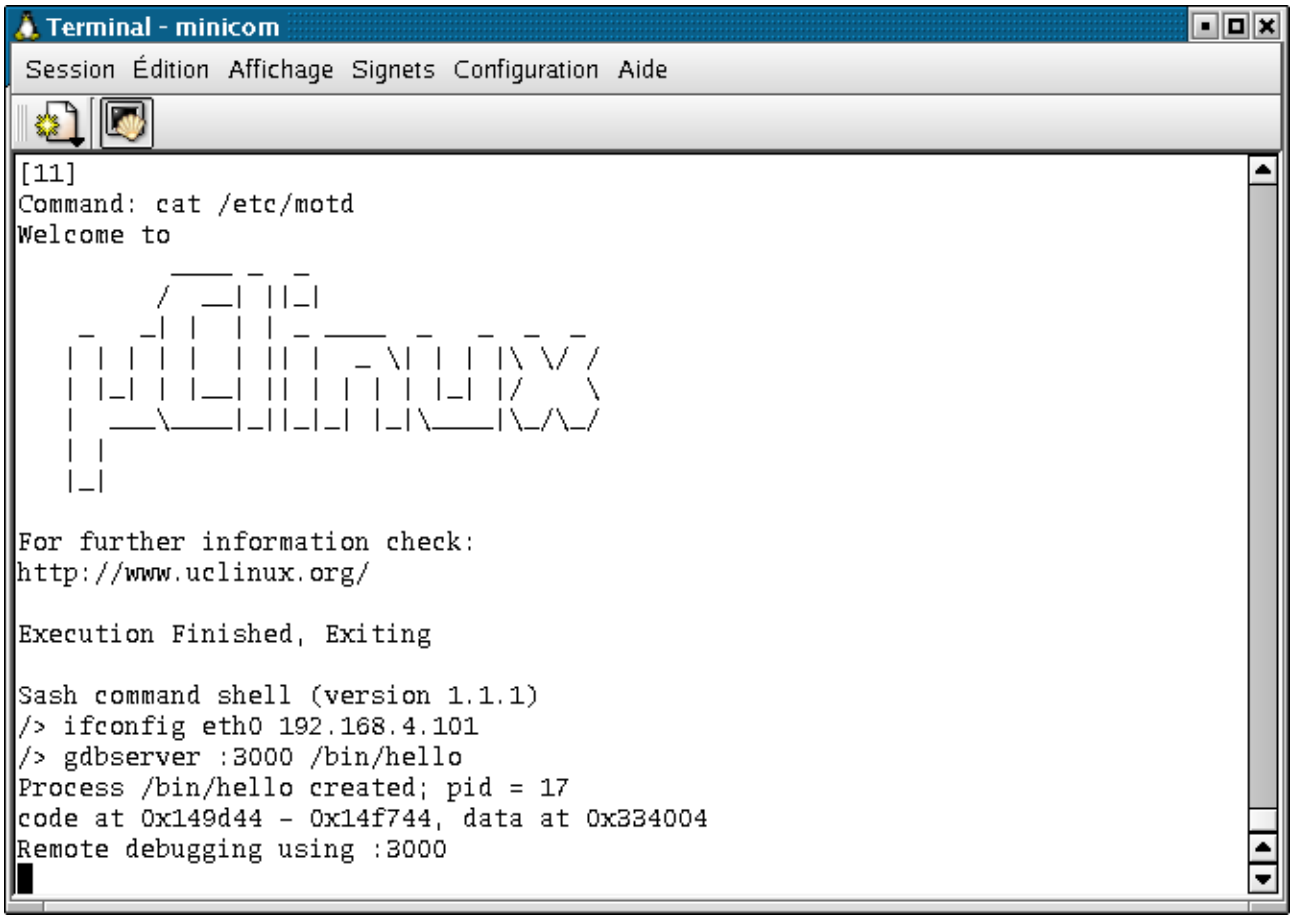


Figure 10 : Session de débogage de l'application µClinux `hello` avec `gdbserver` côté carte cible ColdFire

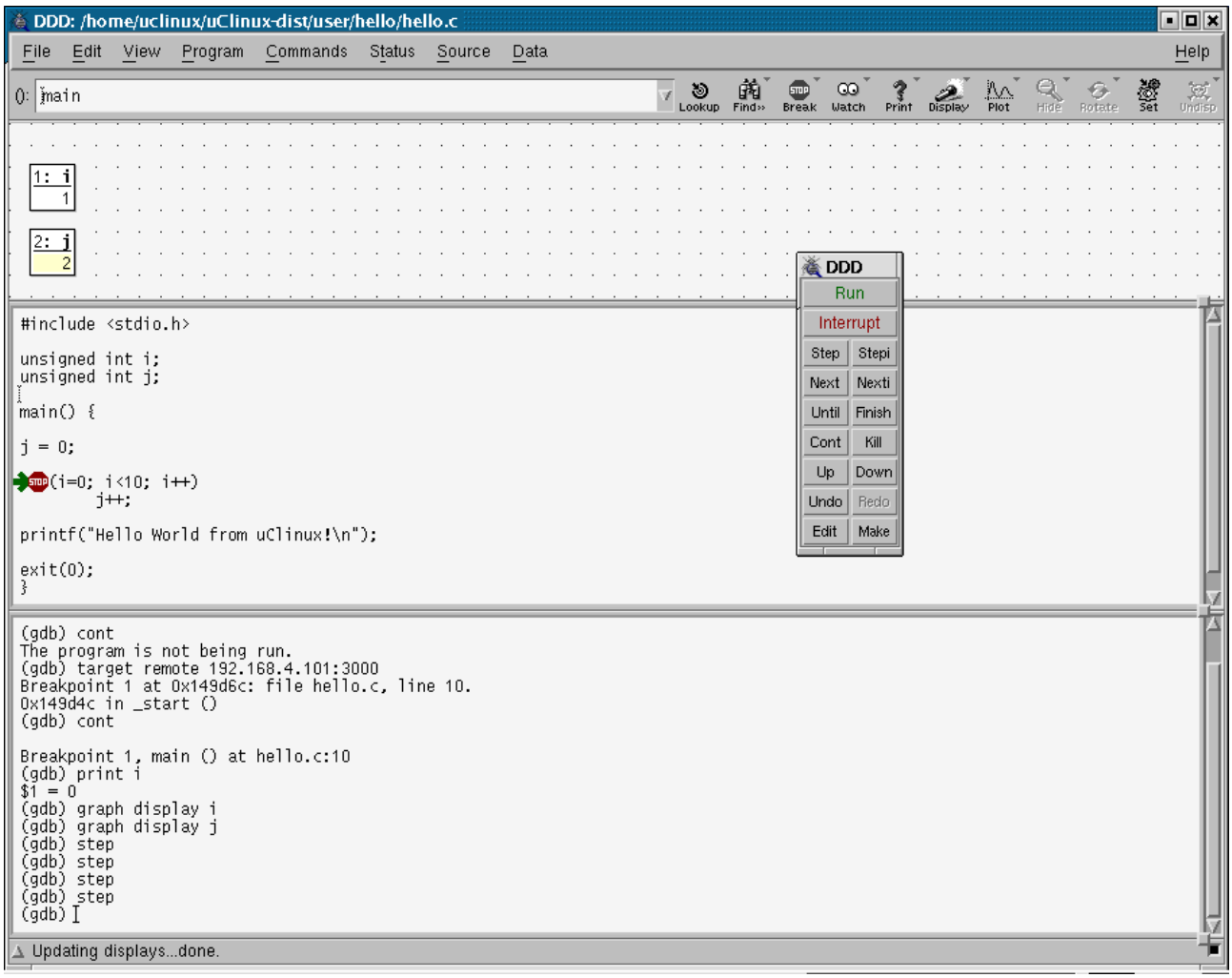


Figure 11 : Session de débogage de l'application µClinux `hello` avec DDD côté PC hôte

Conclusion

A travers cet article d'introduction, nous avons pu voir quels sont les prérequis nécessaires pour « faire » du Linux embarqué. Un exemple concret de mise en oeuvre a aussi été présenté.

Il est d'abord nécessaire d'avoir un environnement de développement croisé. Il est clair que l'on met principalement en oeuvre des outils logiciels libres. Mais vous serez inexorablement obligé d'investir dans du matériel.

On peut bien sûr recycler un vieux PC avec son vieux processeur x86 pour cela mais il est clair qu'il nettement plus « palpitant » de travailler sur les processeurs usuels dans l'embarqué. Vous pouvez vous orienter vers les matériels cités dans cet article ou bien vers d'autres : tout est question de budget. Toutefois, un investissement de 200 € au minimum est à envisager, ou alors, si vous êtes chanceux, vous pouvez essayer de gagner une carte ARM lors du concours organisé avec ce Hors Série ! Bonne lecture et bonne expérimentation sous Linux embarqué !

Bibliographie

- Linux embarqué : le projet µClinux. P. Kadionik. Linux Magazine numéro 36. Février 2002
- Le site de µClinux : http://www.uclinux.org/

- La page sur le portage de µClinux sur processeur ColdFire : http://www.uclinux.org/ports/coldfire/
- La carte cible Freescale M5407C3 de l'exemple : http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=M5407C3&nodeId=018rH3YTLC00M9#
- La rubrique « hardware » de la page Linux embarqué de l'auteur : http://www.enseirb.fr/~kadionik/embedded/embeddedlinux.html#HARDWARE